THE ASSOCIATIVE MODEL OF DATA

SECOND EDITION

SIMON WILLIAMS

LAZY SOFTWARE
The relational model of data, invented by Ted Codd of IBM in the 1960s, is the standard database architecture in use today for mainstream transaction processing and information systems in industry, commerce and government. Relational databases store data in the form of tables (strictly, “relations”) using a separate table for every different type of data. Each table is uniquely structured, so the programs that allow users to interact with the database must be built around the tables, and the structure of the data becomes hard-coded into the programs. This has two consequences. Firstly, each time we develop a new application, we have to create a whole new set of programs to fit the new tables. Secondly, each time we change the structure of the database to store a new type of information, all the programs that use new or altered tables have to be amended and retested.

By contrast, databases that use the associative model of data store all different types of data together in a single, consistent structure that never changes, no matter how many types of data are stored. Information about the structure of the data and the rules that govern it is stored in the database alongside the data itself. This sets the scene for a programming technique called “omnicompetent programming”, whereby the data structures and the rules that govern them are no longer hard-coded into the programs, but are obtained by the programs from the database itself. Omnicompetent programs can work with any associative database without being amended or recompiled in any respect whatsoever. Thus, for the first time, such programs are truly reusable, and no longer need to be amended when the data structures change. This dramatically reduces the cost of application development and maintenance.

Codd’s seminal paper “A Relational Model of Data for Large Shared Data Banks” [4] begins with the words “Future users of large data banks must be protected from having to know
how the data is organized in the machine”. This was a critical step forward in allowing programmers to use their time more productively. But the demands on our limited supply of programming resource are now such that it is time for database architectures to move to the next level of abstraction. The aim of the relational model was to free programmers from having to know the physical structure of data; the aim of the associative model is to free them in addition from having to know its logical structure.

Why is this such an important step? Notwithstanding the efficiency of modern software tools, the development and maintenance of database applications remains extremely labour-intensive, and the cost of building and deploying application software is unsustainably high. The problem is further compounded by the size and complexity of modern applications: SAP’s R/3 ERP system comprises over 16,5001 separate, uniquely structured tables. Such complexity comes at further expense: the implementation costs of major application packages like R/3 are typically between five and ten times the cost of the software itself.

The high cost of software infrastructure is cause enough for concern and action in its own right, going as it does hand-in-hand with the difficulties experienced by small and medium-sized enterprises in gaining access to relevant application software. But the high cost of software is also ultimately damaging to competitiveness. Earlier in my career I spent some years developing and deploying custom-built applications for users of IBM’s System/3X range of computers. The competitive edge that my customers gained from these applications convinced me that enterprises of every size and type can benefit directly and measurably from the use of software that is designed to meet their specific needs and to fit the way they work. Most pre-eminent enterprises become so not by conform-

1 Source: SAP.
ing to established business models, but by radically reinventing
them – would Amazon have achieved what it did if it had tried
to run its web site on someone else’s package? And yet, more
and more companies today deploy the same mission-critical
application software as their competitors, homogenising their
operations and forgoing vital opportunities to gain competitive
advantage. The notion that a successful business should change
the way it works to fit an application package would have been
anathema just a few years ago, but now such compromises are
commonplace. The reason is clear: today, custom-built applic-
ation software is simply beyond the economic reach of all but
those with the deepest pockets.

Why hasn’t the application development industry responded
to this challenge? The answer is that it has been constrained by
the relational model of data. Programming languages have
evolved through abstraction: machine code was abstracted into
symbolic languages; symbolic languages were abstracted into
third generation high level languages. Application development
tool vendors such as Synon, which I founded in 1984, and others
have sought to move to the next level of abstraction through
fourth generation languages (4GLs) but these typically proved
too complex for general usage, and modern object-oriented
programming languages such as Java and C# embody no more
abstraction than their non-OO predecessors. Higher levels of
useful abstraction in programming languages can now be
attained only through a higher level of abstraction in persistent
data structures.

The associative model of data embodies this new level of
data abstraction. It frees programmers from the need to under-
stand either the physical or the logical structure of data, and
allows them to focus solely on the logic and processes by which
users interact with databases, and databases interact with each
other.
Contents

1. LIMITATIONS AND OPPORTUNITIES .......................................................... 1

2. DATA, DATABASES AND DATA MODELS ............................................. 9
   THE EARLY DAYS ................................................................................... 9
   MAGNETIC DISKS .................................................................................. 10
   FUTURE TECHNOLOGIES ...................................................................... 12
   DATABASES ............................................................................................. 13
   METADATA ............................................................................................... 15
   DATA MODELS SO FAR ........................................................................... 17
      The Network Model .............................................................................. 18
      The Hierarchical Model ....................................................................... 19
      The Relational Model .......................................................................... 20
      The Object Model ................................................................................ 25
      The Object/Relational Model .............................................................. 27
      XML ...................................................................................................... 29
   THE RELATIONAL MODEL TODAY ....................................................... 31
      Standards ............................................................................................ 32
      Limitations Again ................................................................................ 33

3. THE RELATIONAL MODEL ..................................................................... 35
   OVERVIEW .............................................................................................. 35
   RELATIONS ............................................................................................ 38
   NORMALISATION ..................................................................................... 40
   RELATIONAL LANGUAGES ................................................................... 42
      Relational Algebra ................................................................................ 43
      SQL ....................................................................................................... 48

4. THE OBJECT MODEL ............................................................................ 49
   OBJECTS AND METHODS ....................................................................... 50
   CLASSES .................................................................................................. 51
   THE OBJECT MODEL OF DATA ........................................................... 53
   SHORTCOMINGS OF THE OBJECT MODEL OF DATA ........................... 55
      Conceptual Model ................................................................................. 55
      Not Everything is an Object ............................................................... 56
      Querying ............................................................................................... 57
      Granularity ........................................................................................... 58
      The Boundary Problem ...................................................................... 59
      Weight and Complexity ...................................................................... 60
5. CRITERIA FOR A NEW DATA MODEL .............................................. 67
   DATA MODELS: A FORMAL PERSPECTIVE ........................................ 67
   MODELS AND MODELLING SYSTEMS .............................................. 70
   PROBLEM DOMAIN ...................................................................... 73

6. INTRODUCING THE ASSOCIATIVE MODEL ............................... 77
   THE BOOKSELLER PROBLEM .......................................................... 79

7. CONCEPTUAL LAYER .................................................................... 83
   ENTITIES AND ASSOCIATIONS ................................................... 83
   Why Associations? ....................................................................... 87
   ATTRIBUTES AS ASSOCIATIONS ................................................. 90
   THE INFORMATION FEATURE ..................................................... 93
   ENTITY TYPES AND ENTITIES ................................................... 93
   Identity ....................................................................................... 94
   Names ....................................................................................... 96
   ASSOCIATION TYPES AND ASSOCIATIONS .................................... 97
   Inverse Associations ..................................................................... 98
   Cardinality .................................................................................. 99
   Default targets .......................................................................... 101
   Inferred Associations ................................................................... 101
   INSTANCE ASSOCIATION TYPES .................................................. 102
   SUBTYPES AND SUPERTYPES .................................................... 102
   SUBSETS AND SUPERSETS .......................................................... 104
   SCALARS, STRINGS AND REFERENCES ....................................... 105
   CREATING ASSOCIATIVE SCHEMAS ......................................... 107

8. LOGICAL LAYER ............................................................................ 109
   ITEMS .......................................................................................... 109
   Identifier ..................................................................................... 109
   Name ......................................................................................... 111
   Item Type ..................................................................................... 113
   LINKS ............................................................................................ 115
   Identifier ..................................................................................... 115
   Source, Verb and Target .............................................................. 115
   Originating transaction ............................................................... 116
   Link Types ................................................................................... 116
   CHAPTERS AND PROFILES ......................................................... 121
UPDATES ........................................................................................................ 124
TRANSACTIONS .......................................................................................... 126
Atomicity .................................................................................................. 128
Consistency ............................................................................................... 129
Isolation ................................................................................................... 129
Durability ............................................................................................... 129
Recovery ................................................................................................. 130
Concurrency .......................................................................................... 130
Security ................................................................................................. 132
Schema Modifications ........................................................................... 132
THE TIME DIMENSION .............................................................................. 133

9. METACODE ............................................................................................. 135
SOFTWARE RE-USE .................................................................................. 136
OMNICHAMPETENT PROGRAMMING .......................................................... 140
SEPARATING RULES AND PROCEDURES .................................................... 143
STANDARD PROGRAM TYPES ..................................................................... 144

10. LANGUAGE FOR THE ASSOCIATIVE MODEL ...................................... 153
A TREE-BASED APPROACH ....................................................................... 153
  Request trees ......................................................................................... 154
  Result Trees .......................................................................................... 156
  Binding Nodes ....................................................................................... 157
  Query Parameters .................................................................................. 161
  Operator Nodes ....................................................................................... 162
  Recursive Closure .................................................................................. 167
A SET-BASED APPROACH .......................................................................... 169
  Associative Algebra ................................................................................. 169
  Querying ............................................................................................... 172
XML: A DOCUMENT-BASED APPROACH ...................................................... 178
  Associative versus Relational ................................................................. 180
GRAPHICAL DESIGN AND MODELLING TECHNIQUES ............................... 181

11. BUSINESS RULES .................................................................................. 185
TERMINOLOGY .......................................................................................... 185
RULES IN AN ENTERPRISE ......................................................................... 186
GOOD RULES ARE DECLARATIVE .............................................................. 187
RULES SO Far ........................................................................................... 189
  Martin and Odell .................................................................................. 189
  Ronald Ross .......................................................................................... 191
  Chris Date ............................................................................................. 193
  Object Management Group ................................................................. 195
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>State and State Domains</td>
<td>195</td>
</tr>
<tr>
<td>State and Database Management Systems</td>
<td>198</td>
</tr>
<tr>
<td>Other views of State</td>
<td>199</td>
</tr>
<tr>
<td>The Components of State</td>
<td>200</td>
</tr>
<tr>
<td>Naming States</td>
<td>200</td>
</tr>
<tr>
<td>Predicates</td>
<td>201</td>
</tr>
<tr>
<td>Effectual and Ineffectual States</td>
<td>202</td>
</tr>
<tr>
<td>Management State Domains</td>
<td>203</td>
</tr>
<tr>
<td>Changes</td>
<td>204</td>
</tr>
<tr>
<td>Rules</td>
<td>204</td>
</tr>
<tr>
<td>Rules in a Database Management System</td>
<td>206</td>
</tr>
<tr>
<td>12. Background to the Associative Model</td>
<td>211</td>
</tr>
<tr>
<td>Historical Perspective</td>
<td>211</td>
</tr>
<tr>
<td>The Associative Model versus the Binary Model</td>
<td>212</td>
</tr>
<tr>
<td>Associative Model versus Binary Model</td>
<td>213</td>
</tr>
<tr>
<td>Associative Model versus Triple Store</td>
<td>217</td>
</tr>
<tr>
<td>13. Challenges of the Associative Model</td>
<td>219</td>
</tr>
<tr>
<td>Abandoning the Record</td>
<td>219</td>
</tr>
<tr>
<td>Distinguishing Entities and Associations</td>
<td>222</td>
</tr>
<tr>
<td>Naming Entities</td>
<td>223</td>
</tr>
<tr>
<td>Using References, Not Values</td>
<td>224</td>
</tr>
<tr>
<td>Moving Away From Object Orientation</td>
<td>229</td>
</tr>
<tr>
<td>Re-asserting the Nature of the Problem Domain</td>
<td>230</td>
</tr>
<tr>
<td>Fewer Programmers?</td>
<td>230</td>
</tr>
<tr>
<td>14. The Relational Model Revisited</td>
<td>233</td>
</tr>
<tr>
<td>Conceptual Foundation</td>
<td>233</td>
</tr>
<tr>
<td>Relationships and Identity</td>
<td>234</td>
</tr>
<tr>
<td>Design Complexity</td>
<td>235</td>
</tr>
<tr>
<td>Implementation Independence Compromised</td>
<td>235</td>
</tr>
<tr>
<td>Sub-optimal Normalisation</td>
<td>236</td>
</tr>
<tr>
<td>Triggers and Stored Procedures</td>
<td>236</td>
</tr>
<tr>
<td>Managing Historic and Future Values</td>
<td>237</td>
</tr>
<tr>
<td>Values and Domains</td>
<td>238</td>
</tr>
<tr>
<td>Modelling Cardinality</td>
<td>239</td>
</tr>
<tr>
<td>Special Values</td>
<td>239</td>
</tr>
<tr>
<td>Nulls</td>
<td>239</td>
</tr>
<tr>
<td>Default Values</td>
<td>240</td>
</tr>
<tr>
<td>Type Values</td>
<td>240</td>
</tr>
<tr>
<td>Web Services</td>
<td>241</td>
</tr>
</tbody>
</table>
15. BENEFITS OF THE ASSOCIATIVE MODEL.................................243

OMNICOMPETENT PROGRAMMING ..................................................243
FEATURE PARTITIONING.....................................................................246
INSTANCE SCHEMA ...........................................................................249
SCHEMA AGGREGATION .....................................................................251
THE TIME DIMENSION .......................................................................255
BUSINESS RULE AUTOMATION..........................................................258

APPENDIX 1: THE BOOKSELLER PROBLEM: SCHEMAS COMPARED

APPENDIX 2: REFERENCES AND BIBLIOGRAPHY

INDEX
1. LIMITATIONS AND OPPORTUNITIES

At the heart of every modern computer system lies a vital software component called a database management system, whose function is to store and retrieve data. Database management systems range in scale from the simple file manager found on every personal computer to highly complex systems spanning many computers, capable of storing huge volumes of data and affording simultaneous access to thousands of users.

Computers didn’t always use database management systems. Early computers stored their data on punched cards, paper tape or magnetic tape. To retrieve a piece of data from part-way through a deck of cards or a reel of tape, the computer had to read all the intervening cards or tape first. By contrast, magnetic disk storage enabled a computer to retrieve any piece of stored data almost instantaneously, and its availability in the early 1960s prompted a flurry of research into how this powerful capability could best be applied.

Three approaches were proposed and developed in the 1960s and early 1970s: the network model, the hierarchical model and the relational model. The relational model won the race to achieve widespread commercial acceptance, and has become the standard used by transaction processing and information systems throughout commerce, industry and government. The network model and the hierarchical model both achieved some commercial success, but then largely fell by the wayside, although implementations of both continue to be used and sold today.

In the late 1980s and early 1990s, the object model of data emerged as the only modern challenger to the relational model. This model first appeared as an adjunct to object-oriented programming languages, in response to the need for computer-aided design applications to store complex diagrams. During the
1990s, the role of object orientation expanded from its origin as a better programming methodology to a guiding philosophy for a wide range of software and business disciplines. Proponents of the object model of data exploited this wave of enthusiasm to build a strong market profile, and for a brief period the object model was widely perceived as the coming successor to the relational model.

Although object-oriented databases achieved some commercial success, they failed to reach a critical mass in the marketplace. Meanwhile, in response to the perceived threat of the object model, many of the established relational database vendors began to incorporate some features of the object model into their relational products. These hybrid systems — often called Universal Servers — are described as object/relational, and some commentators argue that this represents the next major data model. But careful inspection of object/relational databases reveals that they are still fundamentally relational at heart, with some of the features of the object model grafted on.

It is too early to judge whether the market will embrace object/relational technology. To win broad acceptance, its proponents must demonstrate that the marginal value of its object-oriented features justify the cost of its adoption. Given that the market has largely held aloof from the object model of data, it is difficult to see exactly how this might be done. Until the market decides otherwise, the addition of object features to relational products is best regarded as a mid-life kicker for the relational model. Thus the true successor to the relational model has yet to emerge.

The most frequently cited limitation of the relational model is its inability to deal with complex data structures such as text, sounds and images that are typical of multimedia applications. However, the relational model perpetuates six much more fundamental limitations in programming practice that the market has yet to challenge, even though these limitations significantly
increase the cost of computer systems and constrain their ability to enhance competitive edge. These limitations are as follows:

- A relational database uses a separate structure for every different type of data that it contains. Consequently, every new relational database application needs a new set of programs written from scratch, because a program written to use one database design cannot be reused with another. This creates a need for a never-ending supply of new programs, the development and maintenance of which is labour-intensive and expensive. Also, every change to the structure of data in a relational database entails changes to all affected programs, an equally labour-intensive process. Despite their avowed goal of software re-use, techniques such as object orientation, component software and design patterns cannot overcome this ineluctable limitation of the relational model of data. As long as we stay with the relational model, we will continue to spend much more than we should on building and maintaining our software applications.

- Package vendors and web service providers must deliver the richness of functionality required by sophisticated users without overwhelming those with more prosaic needs. The relational model allows software functionality to be tailored to the needs of individual companies through the use of parameters that control the code’s behaviour. This approach is difficult both to develop and to use: over time, as features are added, the increasing complexity of the software becomes a burden to vendors and users alike. Moreover, in a web services environment, the service provider would have to host multiple copies of a package or service to support the differing needs of its customers. Relational technology cannot support customisation of applications and services for individual companies or users from a single, common code base.
• Relational database applications cannot easily record a piece of information about an individual thing that is not relevant to every other thing of the same type, so applications have to store the same information about every customer, order, product and so on. This restriction limits our ability to continually improve the quality of our customer service, because relational database applications cannot record and take account of the needs of individual customers. Until we move on from the relational model, we will not achieve the ultimate in customer service and competitive edge.

• In the relational world, identical types of information in two databases are incompatible: a customer in one database will have different columns in a different order from a customer in another database. Consequently it is difficult, and often impossible, to amalgamate two databases or develop applications that use information from many databases. The potential cost of integrating key systems is a major inhibitor of mergers and acquisitions, and has already forced the abandonment of some significant potential mergers in the financial services industry. Also companies that want to take a more integrated view of their data must incur the extra cost of data warehousing.

• The relational model does not support the declaration and automated enforcement of business rules, because it is prohibitively difficult to implement generic processes such as inference engines over databases whose architecture comprises multiple, disparate structures. Consequently business rules must be hard-coded into programs, and managers have to depend on programmers to determine the behaviour to the applications on which they depend. If the auto industry had evolved in the same way, none of us today would be able to drive our own cars and we would all have to depend on chauffeurs to drive us around.
The relational model is not readily able to record the varying values of attributes through time: in effect, every relational database is a snapshot of data at the instant of the last update, and superseded data is discarded as new values are entered. As we come to better understand the cost of data collection and the value of historic data in improving our understanding of customer behaviour and for other purposes, a database architecture that depends on discarding valuable data as it goes will seem increasingly anachronistic.

Programming languages have evolved through abstraction. Sequences of machine code instructions were abstracted into symbols to create second generation symbolic languages from which machine code could be inferred by an assembler. Sequences of symbols were abstracted into third generation high level language instructions from which machine code could be inferred by a compiler.

In a parallel sense, the first generation of flat file databases required programmers to understand the physical organisation of data, whilst the second generation of relational databases freed them from that requirement. The associative model of data is the first in a third generation of databases that frees programmers from the need to understand either the physical or the logical structure of data, and allows them to focus solely on the processes by which users interact with databases, and databases interact with each other. Using the Associative Model:

- The same set of programs can be used to implement many different applications without being altered, rewritten or recompiled in any way, allowing users to create new applications according to their own requirements by using existing ones as route maps and building blocks. The saving in software development costs afforded by this capability will be substantial.
Applications can permit features to be used or ignored selectively by individual users without the need for code parameterisation. Data sets can be similarly partitioned with precise granularity, to be visible or invisible to individual users. These capabilities support cost-effective customisation of applications to suit the needs of many individual users without compromising core functionality, weakening the integrity of critical data or increasing complexity. This approach is ideally suited to the needs of web service providers and application package vendors alike.

An associative database can record information that is relevant only to one instance of a particular type, without demanding that it be relevant to all other instances of the same type. With this capability, companies can record and take account of the needs of individual customers, and so continue to enhance the quality of their customer service, and thus improve competitive edge, to the ultimate level.

Many databases can be accessed at the same time without extra programming, and databases can be readily merged. Where dissimilar names – perhaps in different languages – have been used for identical types of data or individual data items, users can associate equivalent types and data items to remove ambiguity. This capability of the associative model allows information in different databases to be correlated without the need for data warehousing tools, and permits separate databases to be readily combined.

Business rules may be simply described and stored as part of a database’s schema, removing them entirely from the realm of procedural program code. The associative model’s generic data structure supports the operation of rule engines that can enforce constraints and initiate actions as particular states of data are detected. The ability for managers to specify and
modify business rules directly, without having to depend on programmers to interpret their requirements, is a major step forwards in making computers more user-friendly.

- All changes, including deletions, to an associative database are effected by adding new information, so it is possible to view an associative database as it would appear at any given point in time. This approach acknowledges the high cost of collection and ongoing value of data by providing a database architecture under which no data ever need be thrown away.

The relational model of data is over thirty years old, and its age is starting to tell. It predates the personal computer, object orientation and the Internet, which between them have rewritten almost every rule in the information technology book. Moreover, none of the pressures that fuelled the object model’s challenge have gone away. However, the relational model one of the most firmly established standards in modern computing, buttressed by trillions of dollars’ worth of investment in the applications that lie at the heart of almost every organisation in commerce, industry and government. After the industry’s misguided and short-lived enthusiasm for object database technology, it is not about to try to consign relational technology to the scrap heap again in a hurry.

Yet the associative model still has commercial potential. By way of analogy, consider the microwave oven. The first domestic microwave was introduced in 1967, and was viewed with some scepticism: didn’t every kitchen already have a traditional oven, and wasn’t this new-fangled thing powered by X-rays or some such dangerous stuff? However, notwithstanding the initial reaction, the microwave has progressed from a niche to a mainstream position in the marketplace, to the point where almost every modern British or American kitchen has both a traditional oven and a microwave. Their capabilities overlap to some degree – both will cook a chicken or heat a pie – but there
are tasks to which one or the other is uniquely suited: given the choice, you wouldn’t cook a 25lb turkey in a microwave or a TV dinner in a traditional oven.

Similarly, there is every reason why associative technology is likely to win a niche alongside relational technology. Capabilities will differ: for a year or two, until associative technology has had time to mature, terabyte databases will remain the preserve of relational and older technologies. But where speed of deployment and web friendliness are paramount, or the issues identified above present real barriers to the use of relational technology, associative technology has much to recommend it.
2. DATA, DATABASES AND DATA MODELS

The Early Days

Early computers did not have database management systems based on magnetic disks as we have come to know them. They stored data on punched cards or paper tape. Punched cards date back to the tabulating and sorting machine invented by Herman Hollerith in 1890 to automate the US census. Cards came in decks not unlike decks of playing cards, but with many more cards: sometimes several thousands. The commonest type of card had 80 columns running along its length. Another, smaller type peculiar to some IBM mini-computers had 96 columns arranged in three rows of 32.

Data was recorded on the card by punching a pattern of holes in each column. One pattern would represent the letter “A”, another “B”, another the digit “1” and so on. To understand the questions that the data answered, it was necessary to know how the programmer had decided to use the columns: perhaps columns 1 to 20 for the name of a customer, columns 21 to 30 for the customer’s outstanding balance, and so on. Needless to say, even when writing similar programs, no two programmers would ever choose the same scheme.

Paper tape came in long reels about an inch wide. Data was recorded on the tape in much the same way as punched cards, with holes punched in columns across the width of the tape. Because, unlike cards, the tape had no natural breaks, a special combination of holes would be used to signal the end of one piece of data and the start of another.

Data was read by passing the cards or paper tape between a strip of conductors and a set of wire brushes. The holes allowed the brushes to touch the conductors and pass an electric current, enabling the computer to detect where the holes had been
punched and thus which character had been stored. Later, the wire brushes were replaced by light shining through the holes onto a row of light-sensitive devices.

Punched cards and paper tape were eventually superseded by magnetic tape. Instead of holes, data on magnetic tape was recorded as tiny magnetised dots across the tape’s width. Data was written and read by tiny electric coils called read/write heads. When reading data, a magnetic dot moving past a read/write head induces a current in the coil; when writing data, a current passed through the coil creates or destroys a magnetic dot as it passes the read/write head. In other respects, apart from its increased speed and storage capacity, magnetic tape was much like cards and paper tape.

All these three media suffered from the same two limitations. Firstly, if the computer was instructed to retrieve a particular piece of data located in the middle of the card deck or the tape, it had to read through all the intervening data to find what it was looking for, rather like having to fast-forward through a videotape to find a particular scene in a film. This could take some time, added to the time already needed for a computer operator to locate the correct card deck or tape and load it into a reading device. This method of accessing data is called “serial processing”. Secondly, if the programmer wanted to process data in any order other than that in which it was stored, the computer first had to sort all the data into the desired sequence. Sorting is hard work for computers, and as the volumes of data grew, computers had to spend more and more of their time sorting data instead of doing real work.

**Magnetic Disks**

The invention by IBM of magnetic disks in the 1950s, and their widespread commercial availability in the 1970s were major landmarks in data storage. A magnetic disk is a metal disc
coated with magnetic film onto which data can be written in patterns of magnetic dots (like the patterns of holes in cards and paper tape) by read/write heads that move radially in and out over the surface of the rapidly rotating disk. The combination of the rapidly rotating disk and the movable read/write heads means that any part of the disk is very rapidly accessible, which overcomes both of the limitations of access time and frequent sorting suffered by earlier media.

Software indexing techniques such as randomising and hashing were developed to determine where to locate a particular piece of data on the disk according to some property of the data itself, called a “key”. So if the computer knows the key of a particular piece of data stored on a magnetic disk, it can retrieve the data itself instantaneously.

Keys are important. If you wanted to store some data about Michael Peters in a computer, it might seem reasonable to file and index it under “Michael Peters” or “Peters, Michael”. This may not be a good idea, for several reasons. Firstly, you might want to store data about another Michael Peters, and their data would then be indistinguishable. Secondly, he might change his name, which would require you to find and alter not only my file, but also all references to him within other files. Thirdly, all the keys in an index usually have to be of the same length, so you would have to assign sufficient space to each index entry to cater for the longest possible name that may come along, which would waste space for most entries and make the retrieval process less efficient.

Instead, most computer systems would index Michael Peters’ file under a meaningless code or number, and allow people to look up the code or number when they needed to know it. Programmers call such codes and numbers “keys”. Examples of keys are bank account numbers, American social security numbers and British National Insurance numbers. (Ten digits is enough to give a unique number to every human who has ever lived, so one wonders why some systems use such impossibly
long code numbers. The answer is often that their designers have incorporated data into the keys themselves.)

A computer’s data storage capacity is a measure of the number of individual characters, or “bytes”, of data that it can store. One byte records a single letter of some alphabet, or a single numeric digit, or a space, punctuation mark or special character. To store this sentence requires 41 bytes. Data storage capacity is measured in gigabytes ($2^{30}$ or about $1.07 \times 10^9$ bytes); terabytes ($2^{40}$ or about $1.1 \times 10^{12}$ bytes); petabytes ($2^{50}$ or about $1.125 \times 10^{15}$ bytes); and exabytes ($2^{60}$ or about $1.15 \times 10^{18}$ bytes). Today, every new personal computer has at least ten gigabytes of data storage capacity, and one gigabyte could store the entire text of this book about 2,500 times over.

**Future Technologies**

Magnetic disks remain the standard for data storage today. The technology has evolved out of sight of its origins, and since 1991 the density at which data can be recorded on a magnetic disk has increased at a rate of 60% per year. IBM’s Travelstar 60GH disk drive for notebook computers is about the size of a deck of playing cards, and stores 60 gigabytes of data, sufficient to cover 10 million sheets of paper. Data is stored on the surface of the disk at a density of 28 gigabits/inch$^2$. (A bit is equivalent to a single magnetic dot, or a single hole in a punched card, and usually eight bits make up a byte.)

A few years ago it was thought that the physical limitation of magnetic storage would be reached at the super-paramagnetic limit, which is the point where the magnetic dots become so small that they are unstable, and are liable to flip their polarity in response to tiny variations of operating conditions. This was thought to be at around 12 gigabits/inch$^2$. However, the Travelstar uses a new magnetic data storage technique called antiferromagnetically-coupled (AFC) media, which uses a three-
atom-thick layer of ruthenium (charmingly called “pixie dust”),
a precious metal similar to platinum, sandwiched between two magnetic layers. AFC media is expected to increase current density limits by four-fold, to surpass 100 gigabits/inch$^2$, a level once thought impossible, so magnetic disks may remain the standard for some time to come. Potential successors include magneto-optical storage, where information is recorded magnetically but written and read by laser, and holographic cubes.

## Databases

The advent of magnetic disks meant that, for the first time, programmers were able to write programs that could directly access any piece of a computer’s stored data almost instantaneously. Charles W. Bachman, a database pioneer, pointed out the significance of this development in his 1973 Turing Award-winning paper “The Programmer as Navigator” [1]. Programmers were about to become, in his words, “full-fledged navigators in an n-dimensional data space”. He forecast that this new-found capability would cause as much anguish among programmers as Copernicus’s heliocentric theory caused among the ancient astronomers and theologians.

Bachman was prescient in recognising the implications of direct access to data. For the first time, people were able to think about data as a resource in its own right, separate and distinct from the programs that created and maintained it, and readily available to any program that might be written to use it. It was a short step from here to begin to consider how data might best be structured to take advantage of this potential. This step marked the birth of the database.

So what is a database and what is a database management system? Let’s hear the experts. In their “Fundamentals of Database Systems” [2], Elmasri and Navathe are admirably
concise: “A database is a collection of related data”, and “A database management system is a collection of programs that enables users to create and maintain a database”.

In “An Introduction to Database Systems” [3], Chris Date says: “A database system is essentially nothing more than a computerised record-keeping system. The database itself can be regarded as a kind of electronic filing cabinet; in other words, it is a repository for a collection of computerised data files”. He goes on to refine his definition by saying “A database consists of some collection of persistent data that is used by the application systems of some given enterprise.”

Elmasri and Navathe also describe some important properties of a database:

- A database represents some aspect of the real world, sometimes called the miniworld or the Universe of Discourse (UoD). Changes to the miniworld are reflected in the database.

- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot be correctly referred to as a database.

- A database is designed, built and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

Two vital and related aspects of a database management system are data independence and implementation independence. Data independence means that the application programs that maintain and interrogate the data in a database should not need to be concerned with how the data is physically stored and organised in the database, but should see only a schematic representation of the data, often called a schema. Modern databases consistently achieve this objective.
Implementation independence means that database management systems should behave consistently across different combinations of operating systems and types of hardware architecture (known as platforms) so that programmers should not need to be concerned with the particular platform on which the database is currently running. Modern databases are often less successful in this respect.

**Metadata**

Data comprises ordered strings of characters, ideograms, pictures, sounds and other sensory impressions. Characters include letters, digits, spaces and punctuation marks. Early computers dealt only in characters. The variety of data that could be stored by computers was greatly extended by the multimedia capability of personal computers. Computers are now able to record and reproduce almost the entire gamut of sensory impressions, including voice, music, film and Braille. I can find little current research into the recording and reproduction of smells and tactile impressions beyond Braille, but as soon as the killer applications emerge, no doubt this problem will be tackled and rapidly solved. But there is a critical difference between the ways that computers today process textual data – characters and ideograms – and non-textual data – pictures, sounds and so on. The difference lies in something called “metadata”.

Information is the meaning conveyed by data. To extract the meaning from data, the recipient needs two things: a general understanding of the language in which the data is expressed, and a specific understanding of those concepts that provide the context for the information. Thus, to extract the meaning of “British Airways flight BA123 from LHR to SFO departs at 13:15 daily”, first you need to understand English, and then you need to know that British Airways is an airline, and that LHR and SFO refer to London’s Heathrow airport and San
Francisco’s International airport. These two things, the language and the context, provide us with the information that we need to enable us to extract information from data, and so in this sense they are information about data. The term for information about data is metadata\(^1\). In other words:

\[
\text{Data + Metadata = Information}
\]

Generally, someone who records information for others to use is careful to ensure that their intended audience has access to the metadata that will enable them to extract the correct information from the recorded data. There are two ways to achieve this: either they restrict themselves to language and context that they know their audience will understand, or they put extra metadata into the information, by defining any specialised languages, concepts or terms that they intend to use.

By contrast, there are some rare occasions when it is important to separate the metadata from the data. One such case is where the information being expressed is secret and is to be coded or encrypted before being sent to its intended recipient. Anyone who intercepts an encrypted message will find that they have data but no metadata, and so will be unable to extract any information. Another case is the type of advertisement where part appears on one page of a periodical, another part on a later page and the whole thing later still, or where part appears one week and another part the next. This type of advertising is effective precisely because we are so unused to seeing data devoid of metadata, and, often against our best instincts, our heightened interest sends us searching for metadata. But by far the commonest case in which data is routinely separated from its metadata occurs in the way computers store data.

\(^1\) The term “metadata” promptly violates the semantic distinction between data and information that I have just established. “Meta-information” would be a better term, but its usage is not generally accepted and it is a cumbersome expression, so metadata it remains.
Computers are machines for processing data: they have no other purpose. They capture, validate, store, retrieve, display, print, analyse, transmit and archive data. A computer without data is like a road without vehicles, or a video player without videos. In most computer systems, data and metadata are explicitly and intentionally separated. The simplest example is a personal computer running Windows, where the name of each file has a three-letter extension that determines its type: “exe” for programs, “doc” for documents, “bmp” for bitmaps and so on. If you remove a file’s extension by renaming it, and then double-click on the file, Windows will ask you how it should open the file. That is because a file without an extension contains data with no metadata, and is essentially useless. The extension is not the metadata, but is the key to the metadata. The metadata itself is built into the program that is used to create and manipulate the file. It is the language in which the data is written, and the only things that understand that language are the program that created the file and the programmer that created the program.

**Data Models So Far**

A scheme for structuring data in databases is called a data model. A data model must perform two essential functions: it must provide a set of abstractions that a systems analyst can use to model the information that a business uses, and it must provide a set of techniques to maintain, index and retrieve data on direct access storage devices. (There’s also a more formal discussion of what data model should be in Chapter 5.)

Five data models have been proposed and commercially exploited since the advent of magnetic disks. (Other models such as the functional data model have been proposed, but have not yet found commercial applications, so are outside my scope.) Three models emerged more or less at the same time during the
1960s: the network model, the hierarchical model and the relational model. The relational model came to dominate, and was not seriously challenged until the object model emerged in the late 1980s. Having failed to achieve critical mass in the marketplace, the object model is now in the process of being subsumed into the relational model to form a new model called the object/relational model. Next, we shall look briefly at each of these five models, before going on to consider the two most significant and widely-adopted – the relational model and the object model – in more depth.

**The Network Model**

In the 1960s, Charles Bachman pioneered the first commercial database management system, Integrated Data Store (IDS) at Honeywell. The Data Base Task Group (DBTG) of the Conference on Data Systems Languages (CODASYL) was established to propose database standards, and in April 1971 it endorsed an approach that relied heavily on IDS. This became known as the CODASYL network model, or the DBTG model, now simply called the network model. Bachman provides a good summary of the approach in his 1973 paper. The network model continued to be developed during the 1970s and 1980s, and in 1986 the American National Standards Institute published a standard for a network language called NDL.

The foremost proponent of the network model was Cullinet, in its day the largest software company in the world, and the first to be publicly quoted on the New York Stock Exchange. In the early 1980s, Cullinet’s IDMS was the most widely used database management system in the world. Cullinet failed to foresee the impact of relational technology on its market, and was acquired by Computer Associates in 1989. IDMS is still marketed by Computer Associates today as CA-IDMS.

In the network model, data is stored in records. Each record consists of a group of data items that describe one thing. For
example, a record might contain data about “Michael Peters”, with data items of first name, family name, address, date of birth, telephone number and so on. Records are classified into record types: the record type of the “Michael Peters” record might be “Person”. The record type conveys the meaning and characteristics of each of the data items in the records belonging to it: in other words, it is metadata.

As well as data about a thing, databases also record relationships between things. For example, as well as my name, address, date of birth and so on, we might want to record the name of my children and parents. These individuals are also Persons in their own right, so they each get a record of their own belonging to the record type Person. Then we need a mechanism to record the relationship between one person’s record and another’s. Under the network model, relationships are recorded by set occurrences, which are themselves classified into set types. Each set type comprises an owning record type and a member record type. A set occurrence associates one record from the owning record type with many records from the member set type. We could create set types of “Children” and “Parents”. In both cases both the owning record type and the member record type would be “Person”.

**The Hierarchical Model**


The hierarchical model focuses on the hierarchical relationships which occur in many types of data. It models data structures as record types, similar to those used in the network model, and parent-child relationship (PCR) types, which describe one-to-many relationships between parent and child
record types. Whereas a record type in the network model may be a member record type of many owning record types, the hierarchical model strictly enforces its hierarchical approach by requiring that each record type is a child of no more than one parent record type. This makes the hierarchical model less relevant in situations where there are many lateral relationships between the things described by record types.

The Relational Model

The relational model of data was first described by Dr Edgar F. “Ted” Codd of the IBM Research Laboratory in San Jose, California. Between 1968 and 1990, Codd published more than 30 technical papers and a book on the relational model, and is acknowledged as its originator. There are three significant landmarks in Codd’s work: his paper entitled “A Relational Model of Data for Large Shared Data Banks” [4] appeared in the Communications of the Association for Computing Machinery (ACM) in 1970; his paper entitled “Extending the Database Relational Model to Capture More Meaning” [5] was presented to the Australian Computer Society in Tasmania during 1979; and his book “The Relational Model for Database Management, Version 2” [6] was published in 1990. Codd refers to his work prior to the 1979 paper as Version 1 of the relational model (RM/V1), to the extended version described in the 1979 paper as RM/T (T for Tasmania) and to the version described in his 1990 book as Version 2, or RM/V2.

The relational model views data as relations. A relation is a table containing a number of rows (called “tuples”, pronounced “tupples”) and columns that obeys certain rules. Each relation stores information about a particular type of thing – customers, products, order and so on – and each tuple in a relation stores information about one such thing – an individual customer, product, order etc. Each column of each tuple holds an atomic piece of data that describes the thing that the tuple represents.
Each tuple is uniquely identified by the values in one or more of its columns that are designated as the relation’s primary key, and an association between, say, an order and a customer is recorded by storing the primary key of the customer in a column of the order’s tuple. The customer column of the order is then called a foreign key. The relational model is discussed in more depth in Chapter 3.

Most of the time, programmers use a mainstream programming language such as C++, Visual Basic, JAVA, COBOL or C to write their programs. When they need to use the facilities of a database, they use a second language to define their data structures, or schemas as they are often called, and to write the bits of their program that read and write data from and to the database. This second language is called a data sub-language. Codd’s 1970 paper did not fully describe a data sub-language, but did contemplate one based on applied predicate calculus. In 1971, Codd described ALPHA, a data sub-language based on relational calculus. Other people found Codd’s approach to language too mathematical, and set about devising their own data sub-languages. Of these, SQL has come to dominate to the almost complete exclusion of any other.

SQL is pronounced either “ess-cue-ell” or “sequel”, and was originally called Structured English Query Language or SEQUEL. It was invented in 1974 by a research group at IBM in Yorktown Heights, N.Y. Its eventual adoption as an ANSI standard enabled it to become the accepted data sub-language for defining and using relational databases.

Few standards are criticised as widely or often as SQL. Codd started the ball rolling: in his words, SQL is “quite weak in its fidelity to the (relational) model.” He goes on to remark that “Numerous features (of the relational model) are not supported at all, and others are supported incorrectly. Each of these inconsistencies can be shown to reduce the usefulness and practicality of SQL.” In Chris Date’s “Relational Database Writings 1989 – 1991” [7], he remarks that “... SQL is a very
difficult language: it suffers from such lack of orthogonality, such lack of functionality, such lack of predictability ... in a word it is so ad hoc ...

Nevertheless, SQL’s dominance is so complete that, in data terms, it is like the air that we breathe, and is sometimes referred to, with a mixture of both affection and disdain, as “inter-galactic data-speak”. Other data sub-languages for the relational model have been proposed, including Query Language (QUEL) and “Query by Example” (QBE), but today the battle is over and SQL has won.

Over the last thirty years many other people have described, interpreted and commented on the relational model, and in particular SQL and its shortcomings. The most prolific and apposite is Chris Date.

At the 1975 ACM conference in Ann Arbor, Michigan, Codd and Bachman each fought their corners in the so-called Great Debate on the respective merits of the network and relational models. By all accounts this added more heat than light to the proceedings. In hindsight, one might conclude that the two sides were actually not so far apart. Both were motivated by the need to provide programmers with a view of data that was independent of implementation considerations, and even today some supposedly relational tools owe more to Bachman’s views than to Codd’s. The root of the conflict was Codd’s rigorous, mathematical approach pitted against Bachman’s more pragmatic view. The debate eventually fizzled out as the relational model gradually outpaced the network model in the commercial world.

Within a few years of the publication of Codd’s 1970 paper, people had begun to build relational database management systems. Three initiatives stand out. In the IBM San Jose laboratory, a team led by Frank King began work on System R; at the University of California in Berkeley, a group of students under the direction of Michael Stonebraker and Eugene Wong were building the Interactive Graphics and Retrieval System
(INGRES); and Larry Ellison’s Relation Software Inc was developing the Oracle database. All the major relational databases in the market today, including IBM’s DB2, CA’s INGRES II, Sybase Adaptive Server and Microsoft’s SQL Server, can trace their origins back to one or the other of these projects.

From a database perspective, IBM’s System/38 provides an interesting historical sidebar. Announced in 1978, the System/38’s operating system, CPF, included a built-in relational database management system. Frank Soltis, its architect, claims that System/38 was the first commercial implementation of the relational model (albeit lacking a join capability), predating IBM’s announcement of DB2 by some three years. The System/38’s architecture and database lives on in the shape of the IBM AS/400 (or iSeries as of 2001), a very capable transaction processing system that has never been marketed effectively.

Most commercial vendors produced only partial implementations of RM/V1, and then, faced with the practical demands of the marketplace, pursued their own technical agendas without reference to the master. In particular, several implementations allow the creation of duplicate tuples, which was prohibited in the strongest possible terms by Codd\textsuperscript{1}. In later developments the conceptual gulf between Codd’s vision and its commercial expression grew, leading Codd in 1990 to observe, with palpable bitterness, that “vendors of DBMS products have in many cases failed to understand the first version of RM/V1, let alone RM/T.” Codd himself is at least partly to blame. A mathematician by training, his pedantic assertion of numerous rules, both prescriptive and proscriptive, do not endear him to most readers. In the preface to his book, Codd claims to follow Einstein’s dictum to “make it as simple as possible, but no simpler”, whilst noting

\textsuperscript{1} This may sound like a minor infraction, but in fact the prohibition is fundamental to the relational model. If duplicate tuples occur, foreign keys cannot be reliably de-referenced, and data integrity is compromised.
a few paragraphs later that RM/V2 consists of 333 distinct features. He also recounts his disappointment that the database vendors were not progressing towards full implementation of his 333 features nearly as rapidly as he had expected.

Codd levelled a shot across the bows of the commercial database community with his 1985 Computerworld article entitled “How Relational is Your Database Management System?”, which set out twelve rules that every database claiming to be relational should obey. This list was used from time to time as marketing collateral by one vendor or another to score points against its competitors, but, one suspects, it was never studied too closely in the vendors’ R&D departments.

Codd and the commercial world parted company early on, and were never re-united. One can only wonder how he views the vast personal fortunes made by Larry Ellison, founder of Oracle, and others who have so successfully exploited his brainchild.

The rise of Oracle is a fascinating study. In the early 1980s, Cullinet dominated the database landscape with IDMS, its network database management system, which ran on mainframe computers. Oracle, by contrast, used the relational model and ran on mini-computers such as DEC’s VAX. Oracle was able to win business by offering rapid, cost-effective solutions for departmental users whose needs were a relatively low priority for the glasshouse IT departments who ran the mainframes.

Having won the day against the hierarchical and network models, the relational model has gone from strength to strength since the late 1980s. Specialist companies such as Oracle, Sybase, Informix and Ingres enjoyed phenomenal growth, whilst industry giants such as IBM and later Microsoft came to see that the relational database market was too large and influential to ignore, and introduced their own products. Everything in the relational garden was rosy until the fourth major model of data, the Object Model, came along in the 1990s.
The Object Model

Object orientation evolved during the 1980s as a better way of programming, superseding the prevailing philosophy of structured programming. The core object-oriented concepts of objects and classes first appeared in the programming language Simula 67 in the late 1960s. Soon after, Xerox’s influential Palo Alto research centre developed the first consciously object-oriented language, Smalltalk, which evolved through several dialects and is still in use today, although now largely sidelined for new developments by the advance of Sun Microsystems’ Java, which owes much to it. ‘C’, the native language of the UNIX platform, was originally developed by Dennis Ritchie to run on the DEC PDP-11, and although not itself object-oriented, it became the foundation on which Bjarne Stroustrup built C++ in 1980 at Bell Laboratories in Murray Hill, N.J. C++ has been the most widely used object-oriented language in recent years. More recently, Java burst onto the scene in 1995 riding the Internet wave, and is winning converts very rapidly.

Object orientation is a conceptual model for programming that ensures the integrity of data in main memory by treating each piece of data, or variable, as if it were in the custody of an object. A program that needs to read or alter a variable may do so only 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 object model of data was originally developed to provide a capability called “persistence” for object-oriented programming languages. Persistence allows objects to be stored on disk in essentially the same form in which they exist in memory. The need for persistence first became evident for object-oriented Computer Aided Design (CAD) applications, and products were quick to appear. Some of the more notable early entrants were GemStone, developed by Servio Logic...
Corporation in 1987, which aimed to provide database support for Smalltalk; IRIS from Hewlett-Packard’s Palo Alto laboratory, which aimed to build an object manager on top of the Allbase relational database and introduced an object-oriented version of SQL called OSQL; and ONTOS from Ontologic, which aimed to provide persistent storage for the C++ language and was released in 1989.

By 1990, object orientation had become the prevailing orthodoxy across the entire spectrum of information technology. How and why this came about, often in the face of reason, is an instructive study in its own right, and we shall look at it more closely in Chapter 4. For now, we will simply note that, 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.

In 1991, the market analyst 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.” [8]. In fact, object-oriented databases in 1995 achieved revenues of only $115 million, about one fifth of Ovum’s forecast, and today the market is static at best.

Early predictions for object databases assumed that they would win market share on two fronts: from relational databases for commercial data processing, and from file systems such as NetWare and DOS/Windows for multimedia. In practice, neither happened. For commercial data processing, the relational vendors have held customers’ attention with new developments such as data warehousing and data mining, whilst object databases continued 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.

Notwithstanding the lack of clear-cut commercial success for object databases, the established relational database vendors
were rattled by the new challenger. Most moved rapidly to add object capabilities to their relational products, and to de-emphasise their relational features. Informix led the charge with the high-profile launch of its Universal Server in 1997, but it misjudged the market’s readiness to adopt its new technology, and the introduction was commercially disastrous – in the first quarter of 1997 its license revenue fell by over 50%. Other firms such as Oracle have proceeded in the same direction but more circumspectly, whilst Computer Associates has kept its relational and object-oriented product lines apart, introducing its object-oriented product Jasmine alongside its established relational product Ingres. All this bodes ill for the smaller, specialist object database vendors who sprung up during the 1990s. Most of those that have not been acquired are moving to re-position themselves as niche technologies closer to the Internet.

The Object/Relational Model

Prior to the associative model, the object/relational model is the most recent model of data to be proposed. 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.

First and foremost, the object/relational model was the relational vendors’ response to the apparent challenge of object databases. Most relational vendors have added to their relational products such features of the object model as they think will satisfy their customers’ supposed desire for things to be object-oriented, and have repackaged their products as “Universal Servers”. This was primarily a defensive strategy against the perceived threat of object database, and not a response to customer needs. Evidence of this comes from Informix, which in 1997 led the charge by re-positioning its relational database as a Universal Server. The market failed to understand and buy into
Informix’s new direction, and the company suffered a disastrous drop in revenues which pushed it into a decline that led to its acquisition by IBM four years later.

Several authors have attempted to be more rigorous in their definitions. In “Object-Relational Databases: The Next Great Wave” [9], Michael Stonebraker, founder of database vendors Ingres (now part of Computer Associates) and Illustra, defined 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” [10].

Date and Darwen reassert the continued relevance of the relational model, which they restate and, within limits, redefine. They propose, inter alia, 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 this book. They even pull off a nice piece of semantic misdirection by using the abbreviation “OO”, firmly ingrained into all our psyches as “object-oriented”, to mean “Other Orthogonal”.

Various authors have rightly stressed the importance of a clean, sturdy and consistent conceptual design as the foundation for any application: in “The Mythical Man-Month” [11], Brooks contends that “…conceptual integrity is the most important consideration in system design”. 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 honourable 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.

Sadly for the relational model, no-one is likely to implement Date and Darwen’s proposals because, in the minds of a large proportion of the marketplace, SQL is indistinguishable from the relational model: it would be impossible to sell the benefits of the relational model without SQL, and it would be impossible to implement Date and Darwen’s proposals without abandoning SQL.

Date and Darwen aside, the selective addition of object-oriented features to products whose hearts and souls are relational owes more to marketing than conceptual rigour, and cannot legitimately be viewed as the birth of a new data model. Inspection of today’s hybrid object/relational database management systems reveals that they are still wholly relational at heart, and many users will testify that the addition of object features to a relational DBMS simply provides a richer set of datatypes: a useful step forwards, but a long way short of being a new data model.

**XML**

XML is the abbreviation and common name for Extensible Markup Language, an open standard for describing data, defined by the World-Wide Web consortium (W3C). Like HTML, XML is a subset of SGML (Standard Generalized Markup Language), an ISO standard meta-language for defining the format of text documents. XML defines data elements on web pages and in documents, and provides a way of identifying and transmitting data. A standard of this type is an essential precursor to the automated exchange of transactions between web sites and businesses. (As I write, the XML landscape is evolving so
rapidly that much of what I say is likely to be out of date by the
time you read it.)

The capability of XML in isolation is easy to overstate. XML is simply a mechanism for recording and communicating
data in textual form. This capability is necessary but not sufficient: before XML starts to really earn its keep, common
languages and vocabularies for metadata are needed. By analogy with the evolution of speech, it is as though we are at the stage
of discovering that we can communicate with each other by making sounds, but have not yet evolved language. Initiatives
such as Ariba’s cXML (Commercial XML), Commerce One’s CBL (Common Business Library) and Microsoft’s BizTalk
Framework are starting to fill the gap. The true significance of XML lies not in any of its own technical capabilities, none of
which are new or particularly sophisticated, but rather in the consensus in the industry that it will form the basis for data
communication via the web.

SOAP (Simple Object Access Protocol) is a message-based
protocol for accessing services on the web that uses XML syntax
to send text commands via HTTP. SOAP is set to become the
standard for invoking web services. UDDI (Universal Desc-
ription, Discovery and Integration) is an XML-based specificat-
ion for a registry of businesses and the web services that they
provide. UDDI is designed to enable software to automatically
find services on the web to provide the information needed to
invoke them. Individual services are described by WSDL (Web
Services Description Language).

The concept of an “XML database” has recently come to
prominence. The term is unfortunate and must be treated with
care, in that it leads people to suppose, wrongly, that they must
make a choice between a relational, associative or other “real”
database on one hand and an XML database on the other. XML
databases are not intended to supplant “real” databases, but to
work in conjunction with them. The primary role for XML
databases is to act as aggregators or staging posts for data
communications framed in XML, typically prior to onward transmission to an application or another database.

XML is simply a grammar, not even a language, and is not equipped in any sense to become the core architecture for a database management system. This is the role of a data model, which must provide a set of abstractions to model information, and a set of operations to maintain, index and retrieve data. XML provides none of these, and a collection of XML documents is not a database. It is conceivable that a data model of a sort could be retro-fitted to XML, and that a collection of XML documents could be indexed and cross-referenced to perform some of the functions required of a database, but ultimately inferring internal structure from outward appearance is never a good starting point for engineering rigour, and it unlikely that XML databases will be competing for the same territory as “real” databases anytime soon.

On more positive note, as I shall discuss in Chapter 10, the associative model is much better aligned with the needs of XML than the relational model in the way that it structures data. The market’s enthusiasm for XML is a positive and welcome development, provided that we don’t that lose sight of the fact that most of the work is still to be done.

**The Relational Model Today**

Today, the relational model, imperfectly expressed as SQL, is dominant and ubiquitous to the point where virtually every modern data processing application relies on an SQL database. It has survived through a period during which the personal computer and the Internet have between them recast almost every other significant standard. It has fought off its only convincing challenger in modern times, the object model of data, which it is now assimilating. Although its growth on the UNIX platform has slowed in recent years, it has passed the baton to
the Windows NT platform, where its volume growth rates are impressive, albeit at lower prices per unit.

Even as it seemed perhaps to have reached a growth plateau, the relational model is gaining its second wind through a variety of collateral products for data warehousing and data mining, and preparing to forge ahead once more with object/relational hybrids and specialist Internet products. The relational model is now the standard that underpins trillions of dollars’ worth of investment in database software, the applications that use it, and the systems at the heart of which those applications lie. The reliance of such a significant industry on such a narrow foundation puts one in mind of the reliance of the auto industry on the internal combustion engine.

Standards

As consumers, we are entitled to view standards with ambivalence. On one hand, critical mass in an emerging market is often achieved only after standards emerge, so they play a crucial role in bringing the benefits of new technologies to the majority of potential users who prefer not to endure the trials of the bleeding edge. On the other hand, standards exist more often de facto than de jure, and rarely embody either the best or the cheapest solution to a problem. The real cost of a flawed or second-rate standard – in terms of more money spent than might have been, and of opportunities missed – typically becomes evident only after it has been in place for some time.

Once a technology becomes a standard, the layers of investment that it underpins form a powerful rampart to defend it against the challenges of better or cheaper alternatives. (This fascinating process is described by Geoffrey Moore in “The Gorilla Game” [12].) Nevertheless, sooner or later every standard gives way to a challenger whose claim is undeniable and irresistible. The battle may be short and decisive, as it was when engines replaced horses as the primary source of motive
power for vehicles, or long-running and hard-fought, as will be the battle between the internal combustion engine and some cleaner successor. But if you believe that any standard except death and taxes is here for good, try the fifty year test: cast your mind forward fifty years and ask yourself whether it is likely that the standard you have in mind will still prevail.

**Limitations Again**

The weaknesses and limitations of the relational model are growing more evident all the time. Since it was first proposed, networked personal computers 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 personal computer’s multimedia capability. Whilst the relational model has proved well-suited to transaction processing, it cannot manage the complex data structures such as text, image and sound which are typical of multimedia applications. New applications are emerging that are beyond its capability, such as those dealing with spatial and temporal data, and with uncertainty and imprecision.

Furthermore, relational database engines are large, complex pieces of software, and are not well suited to the Internet, which favours lightweight technologies whose components can be distributed across the Internet itself and embedded in browsers and web sites. And finally, there remain the questions posed by the associative model: why must we write new programs for every application, store the same information about every customer, and throw away historic data as we go? And why can’t we tailor applications for individual users, readily merge databases and automate business rules?
3. THE RELATIONAL MODEL

This chapter takes a closer look at the relational model, briefly describing the model’s salient features. (And I do mean briefly: the two feet of shelf-space in my bookshelf devoted to books about the relational model represent just a fraction of the available literature.) The definitive reference is “The Relational Model for Database Management: Version 2” [6] by the model’s originator, Ted Codd. Chris Date’s many works, some with Hugh Darwen, on both the relational model and SQL are lighter and more accessible than Codd.

Overview

A relational database is best visualised as a collection of tables, each of which comprises a number of columns and a number of rows. The tables are called relations, and the rows are called tuples. Each relation records data about a particular type of thing, such as customers, or products, or orders, and each tuple in a relation records data about one individual thing of the appropriate type – one customer, or one product, or one order. The names of each relation’s columns and the types of information that they will each contain are determined when the database is first set up. Thereafter, new tuples may be added and existing tuples may be altered or removed to reflect the changing set of things in the real world about which data is stored.

For example, a company might set up a database to keep track of its employees and the various departments to which they are assigned from time to time. This would need three relations: one for employees, one for departments and one for assignments of employees to departments. They might look like this:
The set of values from which the pieces of data in a column may be chosen is called a domain. The domain for the Employee number column is all proper employee numbers. As well as a domain, each column also has a name that describes its significance as precisely as possible. Column names are not the same thing as domains: each column uses one domain, but each domain may be used by more than one column. For example, the “Date joined” column of the Employee relation and the “Date assigned” column of the Assignments relation both use the domain Date. I use the word “cell” for an individual column of a tuple, although this term is not in general use.

In every relation, one or more columns together are designated to be the primary key. This means that the values in the primary key columns uniquely identify individual tuples, and distinguish each tuple from others in the relation. Each relation
must have exactly one primary key, no more, no less. Any tuple in a database may thus be uniquely identified by the name of the relation in which it is found, and its primary key values.

In the example above, the headings and data in the primary key columns are shown in bold. The Employee relation’s primary key is Employee number; the Department relation’s primary key is Department code, and the Assignments relation’s primary key is the combination of all three of its columns.

A relation may also contain one or more foreign keys. A foreign key is one or more columns in a relation that have the same domains as the primary key columns of another relation, and which are used to hold values that match the primary key values of a tuple in that other relation, thus implying an association between the tuple containing the foreign key and the tuple whose primary key values it contains.

In the example, the headings and data in foreign key columns are shaded. In the Assignments relation, Employee number is a foreign key that matches the primary key of the Employee relation, and Department code is a foreign key that matches the primary key of the Department relation. As you can see, a column may be both a primary key and a foreign key.

Where a column or columns in two relations both use the same domain, the database software can “join” the two relations to produce a third relation. The third relation does not physically exist distinct from the two joined relations, but is assembled row by row from the two other relations and then presented as though it did physically exist. For example, if we want to see employees’ names alongside their assignments we can join the Employee relation to the Assignments relation using the Employee number column that occurs in both relations, and similarly to the Department relation using the Department code columns. The resulting relation would look like this:
### Assignments with names

<table>
<thead>
<tr>
<th>Employee number</th>
<th>First name</th>
<th>Family name</th>
<th>Dept code</th>
<th>Department name</th>
<th>Date assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>David</td>
<td>Jones</td>
<td>DEV</td>
<td>Development</td>
<td>15-Feb-1995</td>
</tr>
<tr>
<td>123</td>
<td>David</td>
<td>Jones</td>
<td>MKT</td>
<td>Marketing</td>
<td>1-Dec-1998</td>
</tr>
<tr>
<td>234</td>
<td>Alice</td>
<td>Johnson</td>
<td>SLS</td>
<td>Sales</td>
<td>4-Mar-1998</td>
</tr>
<tr>
<td>345</td>
<td>Mary</td>
<td>Trent</td>
<td>MKT</td>
<td>Marketing</td>
<td>31-Aug-1990</td>
</tr>
<tr>
<td>345</td>
<td>Mary</td>
<td>Trent</td>
<td>SLS</td>
<td>Sales</td>
<td>5-Jun-1994</td>
</tr>
<tr>
<td>345</td>
<td>Mary</td>
<td>Trent</td>
<td>SPT</td>
<td>Support</td>
<td>18-Oct-1998</td>
</tr>
</tbody>
</table>

### Relations

Many people who hear the term “relational” for the first time assume that it derives from relationships between the things stored in the database. This is a misconception: the relational model is actually named after the mathematical concept of a relation, which has little to do with our intuitive understanding of a relationship. Some people who should know better promulgate this notion. In their book “Oracle 8: The Complete Reference” [13], George Koch, a former Senior VP of Oracle, and Kevin Loney describe two relations that both contain a column called City, and go on to say “Even though the tables are independent, you can easily see that they are related. The city name in one table is related to the city name in the other. This relationship is the basis for the name relational database.”

This is sophistry. In a relational database, the relationship between two tuples in different relations is not recorded explicitly at all, but must be inferred by someone with sufficient knowledge of the database schema. (Indeed Codd himself argues that the absence of explicit relationships is one of the strengths of the relational model.) Certainly in a trivial example you can easily see relationships, but many real-life databases contain thousands of different column names, and have been developed and maintained over many years by many different
programmers, each with their own view on naming conventions. In such cases, relationships are by no means easy to see.

The mathematical definition of a relation is as follows:

- Given sets $S_1$, $S_2$, . . . , $S_n$, $R$ is a relation on these $n$ sets if it is a set of $n$-tuples, the first component of which is drawn from $S_1$, the second component from $S_2$, and so on.

To understand this, visualise $R$ as a table with $n$ columns. The entries in the first column are drawn from a set of values called $S_1$, the entries in the second column are drawn from a set of values called $S_2$, and so on. $R$ is said to be of degree $n$, and an $n$-tuple is a tuple with $n$ columns. Each of the sets $S_1$, $S_2$ . . . $S_n$ is a domain. If $R$ is our Employee relation, then $S_1$ is the set of all possible employee numbers, $S_2$ is the set of all possible first names, $S_3$ is the set of all possible family names, $S_4$ is the set of all possible dates, and $S_5$ is the set of all possible annual salaries. The Employee relation is of degree 5. A relation’s tuples are called its extension; the descriptive information including the name of the relation, its domains and column headings is called its intension.

Not all tables are relations. A relation has some extra properties that distinguish it from a table. In a relation:

- Every tuple has exactly the same set of columns.

- The values in every tuple of a column must be drawn from the column’s domain.

- The ordering of tuples has no significance: there is no such thing as a first tuple or a last tuple, or any concept of one tuple being next to another. The order in which tuples appear in a list is determined at the time that they are retrieved from the database.

- Similarly, the ordering of columns has no significance.
• No two tuples in a relation may have identical values in
every column.

• No two tuples in a relation may have identical primary keys,
and no column that is part of a primary key may be empty
(or “null” to use the proper term).

• All domains must be atomic – that is, they must consist of
one piece of information only, and that piece of information
cannot be meaningfully sub-divided.

Normalisation

The process of designing a relational schema for an application
uses an important technique called “normalisation”, which
involves removing redundant information and repeating groups
of data from relations.

Imagine a sales order form that includes the customer’s
name and address, together with the names and quantities
ordered of a number of product lines. We could build a relation
that allowed each order to be held as a single tuple. It would
need columns for each part of the customer’s name and address,
and then enough columns, each containing a product name and a
quantity, to cater for the maximum possible number of product
lines on one order – say perhaps 20. The columns containing
product names and quantities are then called a repeating group.
This way of recording information is called “un-normalised”.

Un-normalised structures are deficient in two ways. Firstly,
assuming that we expect to receive a number of orders from this
customer over time, we are wasting time and storage space by
recording the customer’s full name and address on every order.
Instead we can create a separate relation for customers, and
record their names, addresses and other details once and once
only. Each customer would also be assigned some form of
unique identifier, such as an account number, as their primary key. Then, instead of the full name and address, each order need only bear the customer’s account number, which will allow the name and address to be retrieved from the customer relation as and when needed. Apart from the saving of storage capacity and time in avoiding the needless duplication of the customer’s details, this approach also means that, when a customer changes any of its details, we need record the change once and once only, and it will then be effective for every order and every other use to which the information is put within the database. The same applies to product names: it is unnecessary to record the product’s full name on each line when a simple product code which functions as the primary key of a product relation can do the job, and provide access to all the product’s details.

The second way in which un-normalised data is deficient concerns repeating groups. Our sales order form has to contain enough space to cater for the largest order that we expect to receive. This means that for the majority of orders, a considerable amount of space is wasted. Also, to process every product line, a program must enter a loop that allows it to check each product column up to the maximum number, one by one. This is bad news for two reasons. From a programming standpoint, loops in programs are prone to errors and are best avoided unless absolutely necessary. Also, because it is rarely safe to assume that once one blank product column has been found, all subsequent product columns will also be blank, the computer has to loop through all the columns, which can be wasteful.

Normalising our sales order would split it into four relations: one for each order’s header information, a second one for each customer, another to contain the product lines from every order, one per tuple, and a fourth to contain product descriptions.

Codd’s 1970 paper describes the process of normalisation in an elegant and succinct form. Subsequently the concept has been
refined by Codd and others, and to date six normal forms have
been described – first, second, third, Boyce/Codd, fourth and
fifth – each one representing a more esoterically defined
reduction of the original un-normalised form of information
found in the real world.

Relational Languages

Codd proposed two ways of working with relations: the
relational algebra and the relational calculus. The relational
algebra defines the set of eight operators that may be used to
transform relations. The relational calculus is a declarative
formal language based on a branch of mathematics called first-
order predicate logic, which deals with the manipulation of
predicates – functions that return values of “true” or “false”.
There are two branches of relational calculus: tuple relational
calculus and domain relational calculus. In both, the notion of
variables is fundamental. In the tuple relational calculus,
variables range over (ie. draw their values from) relations; in the
domain relational calculus, variables range over the values of
domains.

A solution to a particular problem couched in terms of the
relational algebra is procedural: first do this, next do that, and so
on. The relational calculus, on the other hand, allows the
solution to a problem to be described in a declarative or non-
procedural manner: we describe what is to be done rather than
how to do it. In other words, the algebra is prescriptive whilst
the calculus is descriptive. The syntax of the relational calculus
is beyond the scope of this book, but both Date [3] and

Codd devised an algorithm – “Codd’s Reduction
Algorithm” – which shows that an arbitrary expression in the
calculus may be reduced to a semantically equivalent expression
in the algebra, thus demonstrating that the algebra and the
calculus are equal in terms of their expressive power. He expressed the view that the operators of the algebra should not themselves be directly incorporated into a language, and recommended the use of languages based, like the relational calculus, on first-order predicate logic as being more amenable to optimisation.

The precise nature of the relationship between the relational algebra, the two relational calculuses and relational languages is difficult for initiates to grasp. Early on, Codd proposed a language called ALPHA which was never implemented, and latterly Codd seems to wash his hands of language issues: in [6] he devotes a scant page and a half to the role of language in the relational model, preferring that “... my work remain at a very high level of abstraction, leaving it to others to deal with the specific details of usable languages. Thus, the relational model specifies the semantics of these languages, and does not specify the syntax at all.” In [3], Date expresses the view that Codd set out the relational algebra as the core instruction set for a possible implementation of the relational calculus. ALPHA and the tuple relational calculus strongly influenced QUEL, which was originally developed for INGRES and was for some time a serious competitor to SQL. IBM’s QBE (Query by Example), which was implemented for DB2, was influenced by the domain relational calculus. However, both have been largely superseded by SQL. SQL has its roots in the tuple relational calculus, but SQL is loose and ad hoc whilst relational calculus is formal and systematic.

**Relational Algebra**

The relational algebra is the set of operators that may be used on relations to produce other relations. The purpose of the relational algebra is to allow the creation of expressions that may in turn form the basis for many different functions within a relational database, including but not limited to the retrieval of data.
Codd defined the eight original operators, which are the four set operations of union, intersection, difference and Cartesian product, and the four special relational operations of restrict, project, join and divide.

- The **union** of two relations forms a third relation containing all the tuples that appear in either or both of the two relations.

- The **intersection** of two relations forms a third relation containing only the tuples that appear in both the two relations.

- The **difference** of two relations forms a third relation containing only the tuples that appear in the first and not the second relation.

- The **Cartesian product** of two relations forms a third relation containing all possible combinations of one tuple from the first relation together with one tuple from the second relation. (In RM/V2, Codd de-emphasised the significance of the Cartesian product.)

- **Restrict** forms a relation containing a subset of the tuples in the original relation. (Restrict is also called Select, but this should not be confused with the SQL SELECT operator.)

- **Project** forms a relation containing a subset of the columns in the original relation.

- The **join** of two relations forms a third relation containing tuples that are a combination of two tuples, one from each relation, such that the two tuples share a common value in a specified column of each.
• **Divide** operates on a relation with two columns and a relation with one column, and forms a third relation of one column consisting of all the values in one column of the first relation that are paired with all the values in the single column of the second relation.

Of the eight operators, five – restrict, project, product, union and difference – are primitive, whilst the remaining three – join, intersection and divide – can be defined in terms of the five primitives. For example, the equivalent of join can be achieved by a product followed by a restrict and a project. Inclusion of the three non-primitives is justified for convenience and brevity. Often achieving equivalent results without them is often a non-trivial exercise.

Also, of the eight operations, two (restrict, project) operate on a single relation to form another: the other six all operate on a pair of relations to form a third. The relational algebra thus has the important property of closure, in that the output from one operation may be used as the input to another. This means it is possible to write nested expressions of arbitrary complexity.

Finally, of the eight operators, three – union, intersection and difference – are applicable only to pairs of relations that are type-compatible, that is, have identical columns based on identical domains.
Various authors have proposed new operators, and four are worthy of mention. As Codd’s original eight operators provided no computational capability, there is a consensus that the functionality provided by extend and summarise is essential. The rename operation allows relations created by successive nested operations to have meaningful column headings. Finally, recursive closure supports useful operations on self-referencing relations that are not otherwise achievable.

- **Extend** forms a relation by adding another column to the original relation whose value is obtained by evaluating a computational expression.

- **Summarise** forms a relation by grouping together tuples that have the same value in each of a given set of columns of the
original relation, and creating one tuple for each such group in the new relation. It has the capability to perform aggregate operations such as SUM and COUNT on designated columns of the grouped tuples and present the result as a column in the new relation.

- **Rename** forms a relation by renaming one or more of the columns of the original relation.

- **Recursive closure** forms a relation by joining a self-referencing relation with itself, taking the result and joining it again with the original relation, as many times as necessary. (For example, if an Employee relation has a column containing the foreign key of an employee’s supervisor, recursive closure is needed to extract all of an employee’s direct or indirect subordinates.)
SQL

Finally, the relational model has its data sub-language, SQL. The SQL standard is the responsibility of the International Organization for Standardization (ISO). The current standard is SQL:1999, informally known as SQL3 prior to its adoption.

SQL is both a data definition language (DDL) in that it supports the definition of relational schema, and a data manipulation language (DML) in that it supports the manipulation of data. It does so both interactively and from within application programs that are written in a host language such as COBOL, PL/I or C++. SQL has three central aspects:

- Data definition, comprising the CREATE, ALTER and DROP operations on TABLE and DOMAIN;
- Data manipulation update operations, comprising INSERT, UPDATE and DELETE;
- Data manipulation retrieval operations, comprising the SQL SELECT operator, which embodies the functionality of the relational algebra: restrict (or select), project, product, union, intersect, difference, join and divide.

SQL is intended for use both interactively, where users type SQL statements directly into a terminal, and in embedded form, where SQL statements are physically embedded within a host language such as C, COBOL or PL/I. While most of the literature gives precedence to the interactive form, in everyday usage today the embedded form predominates.

Although SQL/92 is the current standard database language, no database product supports the whole of SQL/92, and most products support additional capabilities outside and beyond the standard. One should not infer from this summary that SQL is simple: the current ISO standard [14] exceeds 3,000 pages.
Object orientation is first and foremost a way of programming that ensures the integrity of data in the computer’s main memory. In object-oriented programming languages, the purpose of an **object** is to act as the custodian of some data. The object guarantees the integrity of its data by not allowing anything except itself to see or touch the data. Any process that needs to read or change the data must do so by invoking one of the object’s **methods**. A method is a process that the object itself knows how to perform on its data, in a manner designed and tested to guarantee the integrity of the data at all times. This way of hiding data from the outside world to ensure its integrity is called **encapsulation**.

An object’s data items and methods are determined by its **class**. A class is an abstract expression of the characteristics and behaviour of a collection of similar objects. A class may **inherit** data items and methods from other classes. An object is an **instance** of its class, and is said to instantiate the class. Instance is another word for object.

Your bank account is a good real-world analogy for an object. You cannot change anything in your bank account directly: instead you send it messages in the form of cheques, deposits and so on. Your bank account’s methods are “Pay a cheque”, “Receive a deposit”, “Produce a statement” and so on. Your bank account instantiates the class Bank account.

For a mental image of an object, picture an egg. The yolk is the data. The white is the methods themselves, comprising procedural code, and the shell is the interface that the object presents to the world outside, comprising the names of its methods and their various parameters.
The Associative Model of Data

Objects and Methods

Let’s look more closely at objects and methods. Take a piece of data: August 11\textsuperscript{th} 1999. In a traditional programming environment, this data item would exist in a computer’s memory as a string of digits: “19990811”. (Computers often store dates as year/month/day so that they can be ordered and compared more easily.) Suppose we want a program to calculate the date of the same day in fifty years time. Simple enough – we just add 50 to the year.

But suppose we forget that the date is stored in year/month/day form, and so by mistake tell the program to add 50 to the last four digits, which is where the year would have been if we hadn’t inverted it. The result when we read the data back is “19990861”, or August 61\textsuperscript{st} 1999, very far from the desired result.

In an object oriented environment, the data would be in the custody of an object, and nothing else would be allowed to access the data directly. The object’s data items are “day”, “month” and “year”, but other programs can’t see either their names or their values. All that the object shows to other programs is its methods, which are the things that an object can do. Some of the methods for our date might be:

- Tell me what date you are
- Tell me just your year (or month, or day)
- Tell me what day of the week you are
- Tell me how many days before or after a given date you are
- Add x years to yourself and tell me the result
- Add x months to yourself and tell me the result
- Add x days to yourself and tell me the result

So to calculate a date 50 years on, we have our program invoke the method “Add x years to yourself”, specifying a value for x of 50, and the object gives us the answer “August 11\textsuperscript{th} 2049”. In this object-oriented environment, we simply do not have the
option of trying to do the job ourselves by getting our hands on the data.

This is an elegant way to work, but why is it so significant? To appreciate the full implication, picture yourself as a programmer working in a multi-tasking computing environment, where one computer’s central processor and main memory may be used not only by your own program, but also by many other programs simultaneously. In a traditional programming environment, your data could be overwritten or corrupted not only by your own mistakes, as in our example, but also by the mistakes of any number of other programmers who may have once worked on the programs executing alongside your own. By contrast, in an object-oriented environment, each piece of data is safely encapsulated inside its own custodian object.

Clearly this technique can significantly reduce the number and the impact of programming errors. Software errors are now the main cause of space rocket launch failures, so the potential for savings and disaster avoidance are clear.

One further word on encapsulation. Confusingly, many object-oriented programming languages allow programmers to circumvent encapsulation by allowing objects to have public data items, which can be read and changed by the methods of other objects. The use of public data is a blatant subversion of object orientation, and, as experienced programmers will know, it almost always bites the hand that practices it in the end.

**Classes**

It would be inappropriate to specify the entire list of data items and methods anew for every new date we come across, so instead we define a class called Date that does the job once and once only. Then, each time we need a new date, our program says “Create a new instance of the class **Date** called **Date of eclipse**”. **Date of eclipse** then comes fully equipped with the data
items and methods that it needs to be a **Date**. So the class **Date** is an abstraction of every date we are likely to need in the future, and all future dates are constructed by reference to the class **Date**.

Another way to look at it is that the class **Date** is a machine whose purpose is to create properly-formed dates. When a new class is created, it is equipped by its creator with a full set of methods, including a special method called a constructor whose sole job is to create new instances of the class and ensure that only valid instances may be created. Once a class is written and thoroughly tested, it can safely be made available for use to other programmers.

When we come across a need for a new class, often we already have a class that does part but not all of the job that we need doing. When this is the case, we can use the existing class as a starting point for the new one, and create our new class by adding the extra data items and methods that we need to the existing class. But we must do this carefully. The old class is already in use, so we can’t add the new items directly to it because this would alter its behaviour, and disrupt programs that rely on it. Nor do we want to make a copy of the old class and add the new items to it to form the new one, because if we later find a problem in the old class, we would then have to fix it in two or more places.

Instead, we use a mechanism called inheritance. We define the new class by associating it with the old one, and then specifying the extra items that we need. When the programming language compiler wants to build a complete picture of the new class, it first builds a picture of the old one by referring to its current definition, and then adds the extra items to create the new one. Suppose that as well as the date we also want to record the exact time of the eclipse, 11:11am on August 11\textsuperscript{th} 1999. We can create the new class, **Date and time**, by taking the class **Date** and adding to it data items for hours and minutes, together with
appropriate methods to deal with them. Then we can say “Create a new **Date and Time** called **Date and time of eclipse**”.

**The Object Model of Data**

The object model of data was originally developed to provide persistent storage for object-oriented programming languages. Whilst 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, the memory is cleared and the objects are lost. Persistence is the capability that allows the objects to be stored on disk, so that when the program is stopped and re-started, it can re-load its objects from disk and carry on exactly where it had left off.

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 you want to keep a permanent record of anything you write on the board during your own meeting, you must copy it onto paper before you leave the conference room at the end of your meeting. Object databases were developed to be like the paper onto which the whiteboard’s contents are written.

The need for persistence first became evident for object-oriented Computer Aided Design (CAD) applications. After word processing, spreadsheets and programming tools, CAD was the fourth “killer application” for personal computers. The data used by the first three of these came in the form of strings of text and numbers interspersed with special characters such as carriage returns to show how the data should be presented. These strings could exist just as readily on disk as in main memory, so no special tools were needed to write the data to disk when the program ended. CAD applications were different.
They allowed their users to create complex graphical representations that existed solely as networks of interrelated objects in main memory, and could not easily be converted into strings of characters that could be saved to and reloaded from disk (a process known as “serialisation”). Hence the need for object databases, which could copy the objects between main memory and disk virtually unaltered.

The object model of data is consistent with the concepts that we discussed earlier: it pictures data items in the custody of objects that derive their characteristics from classes. The precise interpretation and implementation of this concept varies somewhat from one object database to another. In general terms, an object database is a piece of software that can copy individual objects or associated sets of objects between main memory and disk storage without needing to decompose (or normalise) the objects into their component data items. An association between two objects in main memory is typically expressed as a pointer from one object to the other. Such pointers resolve directly or indirectly to the physical address of the object in main memory. As objects are copied from main memory to disk, these pointers are replaced by the addresses of the objects on disk, and vice versa – a process called “swizzling”. As the objects are written to disk, the database builds and maintains an index that tells it where to locate individual objects on disk.

The object model of data 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. 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. Nevertheless, the market’s enthusiasm for object orientation has positioned the object model as a challenger and potential successor to the relational model. Despite this, the object model has not been a commercial
success, mainly because its proponents have never made a clear case that its potential benefits outweigh the costs associated with its adoption. In seeking to challenge the relational model on its home turf, the object model of data suffers from a series of shortcomings, both practical and conceptual.

**Shortcomings of the Object Model of Data**

**Conceptual Model**

The object model lacks a single, unified conceptual model, and so it is subject to interpretation by authors and vendors alike. As many authors have pointed out, a sound conceptual model is vital as the foundation of any application, and the lack of one for object database has allowed an anarchic situation to develop. Authors disagree on basic matters of definition and terminology. Two examples: some people define an object unequivocally as an instance of a class whilst others use the term to mean either a class or an instance. Some insist that inheritance refers to the passing of properties from a class to a subclass, whilst others allow it also to mean the passing of data and method definitions from a class to an instance.

Microsoft’s proprietary object technology OLE exemplifies this well. In 1991, Microsoft created a technique that would allow multimedia fragments such as things like spreadsheets and pictures to be embedded inside other multimedia documents, to create “compound documents”. It called this technique “Object Linking and Embedding”, or OLE Version 1. But what Microsoft meant by an object in OLE Version 1 – essentially a multimedia file – was not what C++ meant by an object, or Smalltalk meant by an object, or any one of a number of object databases then on the scene meant by an object. Microsoft has since re-invented OLE by renaming it “Oh-lay”, and has declared it to be an enabling technology for software component
integration. Throughout, Microsoft has used the term object to mean what it understands by an object at the time, which continues to be a moving target.

**Not Everything is an Object**

As a way of structuring programs and managing data in main memory, the object model has the elegance, simplicity and clarity of an approach that is clearly right. But in seeking to apply the model more widely, there is a trap for the unwary: it is all too easy to start viewing things in the world solely as objects. This alters our viewpoint in a dangerous way: instead of using a model to illuminate our view of the real world, we see the real world only in terms of our preferred model, and ignore the features of the real world that do not fit the model.

This in turn has led to some questionable applications of object orientation. It is held by some to be a sound basis for a business process re-engineering methodology, but is it really appropriate to re-engineer a commercial enterprise around a model where individuals and departments jealously guard their own information against all-comers, and respond only to a limited and precisely-defined set of inputs? Also we commonly use object-oriented analysis and design methodologies for applications that are to be implemented using relational databases: again, is there not a fundamental mis-match here?

Software development is all about modelling. Programming languages model the facilities and capabilities of the computers that they motivate, design methodologies model the behaviour of real-world systems, and database schemas model the structure of information in the real world. Within the constraints of current technology, each model must be as close a fit as possible to the reality that it represents, be it computer hardware at one end of the spectrum, or the real world at the other. Here are some of the ways in which the world of object orientation does not match the real world.
Each object belongs to a class whose behaviour and characteristics it retains throughout its life. Things in the real world have a lifecycle, and alter their behaviour and characteristics as they do so. Children become adults; prospects become customers; trees become telegraph poles.

Each object belongs to one class only: it has a single set of properties and methods. Things in the real world have many different aspects depending on who is interacting with them, and a different set of properties and methods for each aspect.

Objects have no volition: they respond to messages. A wholly object-oriented system would never get started because no-one would send the first message. In the real world, individuals and enterprises have volition and act spontaneously.

Objects have a limited and prescribed set of messages to which they can respond. In the real world, the ability of individuals and enterprises to survive and prosper depends on their ability to formulate useful responses to entirely new messages.

Querying

Querying is the process of obtaining information, in the form of answers to specific questions, from a database: “How many customers do we have in Scotland?”; “Which suppliers are able to supply 2-inch steel widgets by Friday?”, and so on. Given that the object model’s primary goal is to ensure the integrity of data through encapsulation, it is perhaps unavoidable that getting at the data is not quite as easy as it might otherwise be. There are three reasons why querying under the object model is less efficient than under other models.
Firstly, the value of an object’s data items can only be ascertained by invoking a method. This entails at a minimum passing process control to the method, fetching the value of the data item from storage, and passing process control back to the requesting program. Even if the extra work entailed in invoking a method is no more than these two control transfers (and, especially in a polymorphic environment, it is often much more than that) it is additional to the work involved in simply fetching the value of the data item from storage. Thus retrieving data in an object-oriented environment is inherently less efficient than the same operation in an environment that doesn’t implement encapsulation.

Secondly, in practice many queries can be answered on the basis of only part of an object’s data. For example, it is not necessary to retrieve entire orders to answer the query “Which orders are outstanding for Customer X?” – it would be sufficient to retrieve only the customer association of each order. However encapsulation does not (or should not) allow us to retrieve less than the entire object, so in many cases we will need to fetch more data than we actually need to answer our query.

Thirdly, because the values of data items are accessible only via methods, there is no opportunity to optimise large, complex queries. Under the relational model, techniques for optimising queries are well-developed. Database management systems are able to assess the various ways in which a query might be executed, and to select the most efficient, based on knowledge about how the data is physically stored. Under the object model, encapsulation specifically prohibits any such knowledge, and thus renders its use impossible.

**Granularity**

Granularity is the ability to deal individually with very small pieces of data. In an object database, reading or writing an object
is unavoidably a more complex operation than reading or writing only its data: if it were not, there would be no justification for object databases in the first place. Because object databases demand that objects are accessed only by invoking stored methods, a relatively process-intensive way to work, 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. But to improve on the relational model in areas such as schema flexibility, querying, versioning, meta-data management and long transactions, more granularity is needed, not less. Also, new types of application, such as those dealing with temporal and spatial data, with pattern recognition and matching, and with uncertainty and imprecision, are already finding that the object model is insufficiently granular.

**The Boundary Problem**

Many of the things about which commercial databases record data have complex structures in their own right, such as, for example, a sales order placed by a major retailer to one of its suppliers, which might comprise:

- one header, containing information about the order itself, such as an order reference, order date, who it was placed by and so on;

- multiple pages, one for each store to which merchandise is to be delivered, containing the store’s delivery address and instructions;

- multiple lines per page, comprising one line
for each different item to be delivered, containing quantities, prices etc.

In a relational database, orders would be stored in three tables: one for order headers; one for order pages with columns for the foreign key of the order header table; and one for order lines with columns for the foreign key of the order page.

In an object database, the designer faces some difficult decisions. Should the order header, the order page and the order line each be individual classes? If so, the nature of the linkages between order pages and their parent order, and between order lines and their parent pages, must be such as will not violate encapsulation. This approach precludes an order page having easy access to its lines or its header, and makes the assemblage of an order more complex than necessary.

Alternatively, should the entire order be a single class, encapsulating a header, multiple pages and multiple lines together as a single object? In this case, the object becomes extremely complex, with many links of many different types to other objects. This approach also precludes being able to retrieve lines, or pages or headers individually without reference to their parents or children.

This is the boundary problem in the context of object databases. None of these issues are insuperable, but they serve to underline an essential conflict in object database: whilst one might argue that object databases are good at managing small numbers of large objects, they are ill-equipped to manage the complex relationships that this entails.

**Weight and Complexity**

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 via the Internet itself and embedded in browsers and web sites.
In adapting itself to compete with the relational model, and to stay in step with emerging object standards such as DCOM and CORBA, 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, object technology is now simply too long-winded and arcane for day-to-day use. This is exemplified by Kraig Brockschmidt’s “Inside OLE” from Microsoft Press [15], which takes 1,194 pages to describe Microsoft’s object technology. Is it really so unreasonable to compare this with the eleven pages of Codd’s original paper on the relational model?

**Maturity**

Despite having been around commercially for over ten years, the object model is not fully mature as a database management system. The lack of a unified conceptual basis means that there are no agreed specifications for features such as security and authorisation, distributed data, concurrency, logical views and ad hoc queries. Also, many object databases lack features such as performance tuning, query optimisation, mirroring and interfaces to transaction monitors.

Originally, such features were never considered necessary for the target market, and subsequently no vendor has gained the critical mass of market share that would both justify and fund the addition of such features to their development schedules.

**OO Languages Don’t Mandate OO Databases**

It is often presumed that object database is a prerequisite if one is using an object-oriented programming language. This is far from the truth.
There are two sorts of objects. Memory-based objects such as program variables and graphical components that typify Windows applications exist primarily in main memory and are seldom, if ever, written to disk. Data-based objects such as individual customers, sales orders, insurance policies and so on exist primarily on disk and are brought into main memory relatively rarely, when they are operated upon, and exist there for a relatively short time. Whilst data-based objects may be viewed as instances of classes, it does not follow that they should be stored in an object database, even when applications are being developed using object-oriented language such as C++ or Java. Requirements in each case differ significantly, and whenever the need for persistent storage arises, the correct solution should be judged on its merits.

Different types of mechanism are needed to protect the integrity of memory-based objects and data-based objects: identity and encapsulation for memory-based objects; properties of atomicity, consistency, isolation and durability (ACID) and user authorities for data-based objects. Also memory-based objects typically have relatively few associations with other objects, so their encapsulation does not represent a significant overhead, whilst data-based objects are typically rich in associations with other objects, so their encapsulation represents a significant overhead to query and other operations.

Moreover, memory-based objects have relatively simple internal structures, so decomposition and normalisation are not relevant, whilst disk-based objects often have complex internal structures, so decomposition and normalisation are important: however decomposition and normalisation are not intuitively consistent with the object model because they violate encapsulation. Finally, very large numbers of memory-based objects do not often co-exist simultaneously, so indexing and other techniques to locate and disambiguate them do not need to be particularly sophisticated, whilst equivalent techniques for
disk-based objects must be capable of rapidly locating and disambiguating tens of millions of similar instances.

There is another consideration. Object oriented techniques can be applied with good effect to the development of relational database applications using non-OO languages such as COBOL or RPG. Synon/2\(^1\) does precisely this. Each time a new table is defined, methods to create, read, update and delete its rows are created automatically. Synon/2’s procedural language contains no explicit database I/O instructions: instead its code generator enforces the use of each table’s create, read, update and delete methods for I/O, which are implemented as subroutines within the generated RPG code. Any development shop may elect to use a similar approach, and thus reap many of the benefits of object orientation without the need for an object database or even an object-oriented programming language.

**Do Object Databases Have a Future?**

After many years without significant commercial success, during which its challenge to the dominance of the relational model has been effectively rebuffed, it is appropriate to ask whether object database has a future.

First, a look back at how the present state of affairs came to be. The object model of data was not originally conceived to improve upon or even to compete with the relational model, but was intended simply to provide persistent storage for object-oriented programs. How then did it come to be regarded as a potential successor to the relational model? Like any other industry, the software business enjoys its fashions, and during the early 1990s object orientation became fashionable. In a

---

\(^1\) Synon/2 is an application generator that my colleagues and I at Synon created in 1985, which went on to become the dominant application development tool for the IBM AS/400 (now iSeries). It is currently marketed by Computer Associates as Advantage 2E.
world of ever more complex “technologies” and three-letter acronyms, its elegance and simplicity struck an almost folksy, back-to-basics note that people find attractive. But this same simplicity meant that object orientation was a blank canvas on which one could paint one’s own interpretation. Inevitably, more of the interpretation came from marketing departments than from development labs. Pre-OO products were hastily repackaged and repositioned as object-oriented, often regardless of whether or not they had a single true object-oriented feature (many did not), and product managers were quick to use their newly minted object-oriented credentials as a competitive weapon against those who were slower or less cynical than themselves. Soon, the ability to demonstrate current or in-plan object-oriented features was a pre-requisite in the software product sales cycle.

In this climate, it would have been unreasonable to expect that the market for database management systems, one of the largest and most influential sectors, should remain unaffected, and indeed it did not. By the early 1990s, market analysts were confidently predicting that the object model would overtake its relational cousin in the marketplace within the decade. Hindsight reveals that they were wrong.

Looking forwards now the froth has subsided, a major determinant of the ultimate size of the market for object database is whether its original objective – to provide persistent storage for object-oriented programming languages – was ever commercially sustainable. With the benefit of hindsight, the answer seems to be no. The obvious early adopters of such technology were software vendors developing personal computer applications such as spreadsheets, word processors, CAD and CASE. If cheap, robust object databases had been more widely available sooner, perhaps enough vendors might have adopted them to create a sustainable market. But this didn’t happen. Vendors were unwilling to depend on suppliers of object technology who had not themselves achieved critical
mass in the marketplace, and lack of a dominant player meant that the market didn’t consolidate quickly enough. Also, hardware became cheaper and much more powerful, so serialisation (the process of converting objects in memory into a character streams that can be written to disk) soon ceased to be the resource-intensive challenge that it had once been.

When a technology first catches the market’s attention, expectation is high, fuelled by the claims of vendors and industry commentators, which are typically based on the anticipated capability of the mature technology, rather than today’s reality. An excess of expectation over reality can lead to early disappointments as pioneers fail to achieve the benefits they had hoped for, and a collapse of market expectation and interest may result. This is the danger zone, when promising but immature technologies may be abandoned almost overnight. However, if the technology has real potential to add value, its true capability will continue to advance, and over time the market’s expectation will come back in line with reality. Today, the object model of data is in the danger zone.
5. CRITERIA FOR A NEW DATA MODEL

My conclusion from the foregoing is that the relational model will not remain the dominant standard for commercial database management systems indefinitely, and that neither the object model nor the object/relational model will supersede it. Before I describe my proposal for one candidate successor, I want to consider the criteria by which all such candidates should be judged. First, I shall examine what a data model should comprise in a more formal way.

Data Models: a Formal Perspective

The general need for data models is asserted by Codd in the opening sentence of [4]: “Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation).” Codd credits his colleague C T Davies of IBM Poughkeepsie with convincing him of the need for data independence.

This theme is reflected in work from the early 1970s by the CODASYL Data Base Task Group (DBTG) and the IBM user groups Guide and Share. In 1975 the ANSI-SPARC committee proposed a more formal three-level meta-model, which consisted of an external level comprising the users’ individual views of the database, a conceptual level comprising an abstract view of the whole database, and an internal level, comprising the physical representation of data in the database. The conceptual level corresponds to the DBTG concept of the “schema”, and the views in the external level corresponds to the DBTG concept of “subschemas”.

Codd returns to the issue more formally in his 1979 paper [5], in which he states that relational model consists of:

1. A collection of time-varying tabular relations;
2. The insert-update-delete rules (ie. the entity integrity rule and the referential integrity rule); and

3. The relational algebra.

Codd also observes that various semantic decomposition concepts are closely associated with (ie. are almost part of) the relational model. He goes on to describe an extended version of the relational model called RM/T in an attempt to capture more of the meaning of the data by embracing concepts dealing with molecular semantics such as aggregation and generalisation.

In Chapter 8 of “Relational Database: Selected Writings” [29], Chris Date presents some of Codd’s work in a more accessible way, and goes on to deal with how the relational model should be interpreted. Referencing Codd, he asserts that a data model consists of a collection of data object types, a collection of general integrity rules and a collection of operators. Date goes on to clarify Codd’s non-exhaustive list of six uses for a data model, and articulates his own interpretation principle, namely that a data model “must have a commonly accepted (and useful) interpretation; that is, its objects, integrity rules and operators must have some generally accepted correspondence to phenomena in the real world.”

Date presents arguments to support his claims, stressing that data models are formal systems, whilst the real world is an informal system, and thus a data model must use formal behaviour to mimic informal aspects of the real world. The rest of Date’s arguments focus on explaining how the relational model should be interpreted to conform to his interpretation principle.

It is interesting to compare and contrast the approaches that Codd and Date have taken to the relational model and, by inference, to data models and their scope more generally. Codd, the mathematician, has grown increasingly rigorous, fine-grained and proscriptive, lamenting that the commercial database world has failed to follow him. Date, the comm-
unicator, has sought through his Relational Database Writings series to interpret, amplify and render accessible most of the more arcane aspects of relational database theory. Date also expresses the belief that “Everyone professionally involved in database management should be thoroughly conversant not only with the relational model per se, but also with its interpretation.”

The purpose of a data model is to provide an abstract view of data and schema in a database, so that its users don’t need to know how the data is physically stored and retrieved. This objective, called implementation independence, is desirable in order to ensure that the programs that maintain the data and the queries that users ask of the data do not need to change as the hardware and software platforms on which the database runs evolve, or as the database is re-implemented on other platforms.

To this end, a data model needs to include, as a minimum, a set of abstractions that database designers may use to represent types of things in the real world and their properties, together with the integrity rules that govern how instances of those abstractions may interrelate, and the types of operations that may be performed on such instances. A proposal may qualify as a data model based solely on this (the relational model did) but today its credentials are strengthened if it also includes at least one proposal for a set of physical data storage constructs that may be used to implement a database management system based on the data model, together with appropriate mappings between the physical constructs and the abstractions.

It is also useful if a data model addresses how it proposes to fulfil the many practical demands that the modern data processing environment imposes, such as how data may be secured, transmitted and distributed.
Models and Modelling Systems

In view of their function, data models might be more appropriately called data modelling systems, and in fact they are both modelling systems and models: modelling systems in the sense of providing a framework for modelling information in the real world, and models in the sense that they themselves model the structure of information in the real world.

A model is a representation of something in the real world, created for a specific purpose, whose function is to enhance our perception and understanding of a particular aspect of its subject. Consequently a model emphasises certain features of its subject, and ignores or de-emphasises others. For example, a road map is a model of the terrain that it portrays. It emphasises roads and towns, and ignores or de-emphasises the rural and urban landscape through which the roads pass.

A modelling system, by contrast, is a set of components and tools for building models, together with rules about how the components should be used. A modelling system for road maps might include blue lines for motorways, green lines for major roads, grey shading for towns and so on. (In object-oriented terminology, the modelling system is the class; the model is the instance.)

The purpose of a data modelling system is to model the operational, commercial, financial and legal relationships and transactions within and between organisations, and between organisations and people. That model then becomes the blueprint for an application. We shall work backwards from applications to modelling systems to consider the factors that influence quality all along the line.

What makes a good application? Well, first, what do we mean by “good”? The answer is twofold: firstly, a good application must fulfil its users’ needs now, and secondly it must be capable of adapting so that it continues to fulfil those needs as they change over time. Conceptually an application comprises
two parts: a data model and a user interface. Broadly, the user interface determines how well the application meets its users’ needs now, and the data model determines how well it can adapt over time. It is now generally acknowledged that the lion’s share of software development budgets are spent on maintenance rather than new development, so from an economic standpoint the ability to adapt over time (or more fully stated, the ability to be maintained at reasonable cost) is clearly the more important criterion. Thus the data model is more important than the user interface in determining the quality of an application. An application with a good data model and a poor user interface is not necessarily a good application, but an application with a poor data model can never be a good application. A good data model with a poor user interface is preferable to a poor data model with a good user interface, because it is relatively easy to fix a poor user interface and almost impossible to fix a poor data model.

Moreover, a user interface can be inferred from the data model, but a data model (other than the most trivial) cannot be inferred from a user interface. This debate was current in the 1990s, after Microsoft’s Visual Basic introduced a new generation of application development tools that encouraged a “glass-in” approach: in other words, the application designer should begin by designing the user interface, and from that should infer the data model. It is now generally acknowledged that this approach works (or, more correctly, does not usually fail) for small applications involving up to say, ten relations, but is not well-suited for going up-scale to address applications with, say, twenty-five relations or more.

To put the subject in perspective, most non-trivial departmental systems start at around fifty relations; an enterprise-wide financial system will usually comprise several hundred relations, and a full Enterprise Resource Planning (ERP) system will often comprise several thousand relations. To attempt to construct applications on this scale without the benefit
of a data model is rather like trying to construct an office building without the benefit of a plan: the user interface is a refinement of a data model, and not vice versa. The data model is the foundation on which an application is built.

The true test of an application is not how it performs during its first few months, but how it performs after about a year. By then, its users will have found ways around any features of the application that are at odds with reality. One of the most sobering and informative experiences that an application designer can have is to revisit one of their creations after a year or so, and see for themselves just how much of it is being used as they had intended, and how much of it has been subverted by users who have to get the job done regardless.

Good systems in the real world (that is, non-computerised systems) are rarely static – they evolve continually in small ways as they respond to changing operational conditions. The current generation of applications tend to oppose this evolutionary pressure, because even the good ones are inherently rigid. Notwithstanding this, applications whose data models are a good fit to the real world are more malleable and better able to adapt to small changes, whilst those whose data models are not a good fit will work for a while, but will be brittle in their response to evolutionary pressure: small changes will prove difficult or impossible, so the application soon becomes a burden on its users and they begin to work around it in ways that can seriously affect the quality of information that it provides.

So a good data model is the primary determinant of the quality of an application, and a data model is only as good as the modelling system that creates it. Every data modelling system operates within a set of constraints that include the typical skill level of modellers, the cost of the modelling process, and, most significantly, the available technology. Looking back at the programs I was writing in the 1960s, most of my code did no useful work for the company that employed me, but was written to overcome the limitations of the technology with which I was
working: assembler-level programming languages, minimal main memory and storage media capable only of serial processing. The best modelling system is the one whose models are the most consistently true-to-life as possible given the constraints within which it operates. This is why the relational model succeeded against the network and hierarchical models: the models that it produced were simply more true to life, and offered a better fit in more diverse situations.

As technology advances, constraints are relaxed, and modelling systems must change in response: a modelling system may compromise the integrity of its models only to the extent forced upon it by implementation technology, and no more. Each time technology passes a significant milestone, modelling systems must adapt or be superseded. The advent of direct access to data through disk storage was such a milestone, and it rendered the serial access model of data almost irrelevant. Since the availability of the first commercial relational databases, the cost per megabyte of storage has decreased by a factor of more than 250,000, and the speed of access to data has increased significantly. This constitutes another such landmark, to which we must respond by updating our modelling systems.

**Problem Domain**

A primary characteristic of any modelling system is the problem domain that it addresses. The problem domain is that aspect of the real world with which we are concerned for some purpose, sometimes also called the miniworld, or the Universe of Discourse.

To establish a problem domain we divide the world in three orthogonal directions. First we divide it by deciding that we are interested only in certain types of things. Then we divide it by including only certain aspects of each type of thing and excluding others. Finally we divide it by including only some
instances of each type of thing that we are interested in and excluding others of the same type. For example, if we were making a road map of part of England, we might decide that we were interested in roads, ferries, towns, rivers and coastlines, but not buildings, terrain and other natural features. Then we might decide that we were interested in the length and direction of roads and rivers, but not their width. Lastly we might decide that we were interested in London and the South East, but not Bristol and the South West.

The first two cuts usually fall within the preserve of the modelling system, whilst the third usually falls within the preserve of the model. This means that a modelling system must be capable of dealing with any and every instance of the types of thing that it addresses.

The question of which problem domain should be addressed by the next generation of commercial database management systems is fundamentally important, and the answer is perhaps not an obvious one.

The mainstream database industry made the assumption around the middle of the 1990s that the problem domain for commercial databases should be extended to include “objects”. There were three forces driving this assumption. Firstly, the relational model’s inability to deal with multimedia data is one of its more obvious shortcomings. Secondly, objects are fashionable. Thirdly, the relational database vendors believed for a while that their market was under threat from object databases.

In hindsight, it is beginning to look as though this assumption was ill-informed. The line of reasoning that led to it went something like this: objects are fashionable; multimedia files are objects; there are more multimedia files than there used to be; therefore the market needs object databases to store multimedia files. But multimedia files are not objects in the object oriented sense (OO-objects). True, they have identity, state and behaviour, but so do plenty of things that are not OO-objects – and moreover, there are simply no benefits to be
gained from treating multimedia files as OO-objects, or from storing them in object databases.

Take, for example, a Microsoft Word document. The single valid OO-class for such an object is Word itself, and the construction of an alternative by anyone other than Microsoft would infringe its intellectual property. A Word document can readily be identified by a path name and a file name, or a URL. Associations with other objects are simple in nature, few in number, adequately managed by Microsoft’s OLE technology, and cannot be managed in any other way without replicating major chunks of OLE. Mechanisms already exist within Word itself to decompose a document into smaller objects and to manage multiple versions. At the same time, vendors are investing in directory services based on Lightweight Directory Access Protocol (LDAP), which promise over time to substantially improve the way in which resources of all types are located across networks and over the Internet. Given all this, it is difficult to visualise exactly what extra benefits are expected to accrue from keeping multimedia files inside an object database.

So what about other sorts of objects? As we have observed, object databases were originally conceived to provide persistent storage for OO-objects inside object-oriented programming languages. But this is precisely the market that object databases have been attacking. If specialised products have failed to establish a sustainable market after ten years, it is most unlikely that such a market will develop around such facilities when they form only a part of much more generalised, “universal” database products.

Then what about new types of application all together? These include applications dealing with temporal and spatial data, with pattern recognition and matching, and with uncertainty and imprecision, as well as those that address specialised multimedia needs such as modelling, structuring, content addressability, searching and long transactions. But these are all highly specialised requirements at an early stage of
development, and in the light of continued software industry consolidation, the potential customer base is unlikely to warrant the inclusion of facilities directed solely to supporting such requirements in the next generation of general-purpose database management systems.

In the light of all this, we are led to conclude that, despite all the attention being paid to multimedia and other specialised forms of data object, the problem domain that ought to be addressed by the relational model’s successor is the same one that Codd addressed in 1970; namely information about the operational, commercial, financial and legal relationships and transactions within and between organisations and between organisations and people. This type of information is predominantly character-based.

Whilst such information can usefully be illuminated with multimedia content, such content is and will continue to be secondary and derivative. The need to store such data side-by-side with character data is a technically trivial requirement that can be (and in most implementations already is) adequately addressed within the relational model by the inclusion of datatypes for large objects, and of pointers to external files.

The real opportunity for the next generation of database management systems lies not in finding better ways to store and index web sites, but in using our vastly increased resources of data storage capacity and access speed to improve our capability to store and query our mission-critical enterprise and transactional data.
6. INTRODUCING THE ASSOCIATIVE MODEL

Before we look at the theoretical detail of the associative model in the following chapters, this chapter introduces the basic idea by means of two examples, with the minimum of theory.

In the associative model, a database comprises two types of data structures:

- **Items**, each of which has a unique identifier, a name and a type.

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

(The proper word for the connecting part of a sentence is “copula”, but I use “verb” which is more intuitive but slightly less accurate. Not all associative model “verbs” are actually verbs; many, such as “at” or “on” are abbreviated to prepositions.)

Let us see how the associative model would use these two structures to store the piece of information “Flight BA1234 arrived at Heathrow Airport on 12-Aug-98 at 10:25am”. There are seven items: the four nouns **Flight BA1234, Heathrow Airport, 12-Aug-98** and **10:25am**, and the three verbs **arrived at, on** and **at**. We need three links to store the data. They are:

**Flight BA1234** arrived at **Heathrow Airport**
... on **12-Aug-98**
... at **10:25am**
The first link is the first line. It uses the verb **arrived at** to associate the items **Flight BA1234** and **Heathrow Airport**. The second link is the first and second lines combined. It uses the verb **on** to associate the first link and the item **12-Aug-98**. (A link that begins with an ellipsis “...” has the previous link as its source.) The third link comprises all three lines. It uses the verb **at** to associate the second link and the item **10:25am**.

Sometimes when writing links, instead of using new lines to show each link it is more convenient to keep going in a long string. When we do this, we simply put brackets around each link. Written this way, our example would look like this:

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

If we see the associative model through the eyes of the relational model, we can store an associative database in just two tables: one for items and one for links. We give each item and link a meaningless number as an identifier, to act as its primary key.

<table>
<thead>
<tr>
<th>Items</th>
<th>Identifier</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>Flight BA1234</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>Heathrow Airport</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>12-Aug-1998</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>10:25am</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>arrived at</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>at</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Links</th>
<th>Identifier</th>
<th>Source</th>
<th>Verb</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>Source</td>
<td>Verb</td>
<td>Target</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>77</td>
<td>12</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>74</td>
<td>67</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>03</td>
<td>09</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

(These are not true relations, because each of the entries in the Source, Verb and Target columns may be an identifier of either an item or a link, and the Name column may contain almost
anything at all. This sort of ambivalence is not permitted by the relational model.)

The Bookseller Problem

Now let us look at a more sophisticated problem that shows metadata as well as data. In the context of a particular problem, the metadata that expresses the solution is called a “schema”. First, here is the problem domain:

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 is the schema that describes the structure of orders. The items in bold are entity types: we shall discuss exactly what that means later.

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 itself. First, the entities that we are dealing with:

Amazon is a Legal entity
Bookpages is a Legal entity
Dr No is a Book
Michael Peters is a Person
Michael Peters 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-98
... at £10
... in America
... from 1-Mar-98
... at $16

Amazon sells Spycatcher
... worth 50 points
... in Britain
... from 1-Jun-98
... at £7
... in America
... from 1-Jun-98
... at $12

Bookpages sells Dr No
... worth 75 points
... in Britain
... from 1-Jan-98
... at £8
... in America
... from 1-Jan-98
... at $14

Bookpages sells Spycatcher
... worth 35 points
... in America
... from 1-Jun-98
... at $13

Now, for each of our two customers:

Michael Peters customer of Bookpages
... has earned 1,200 points
... orders Dr No
... on 10-Oct-98
... at £10
Mary Davis customer of Amazon
... has earned 750 points
... orders Spycatcher
... on 19-Oct-98
... at $12

Here is the metadata for the bookseller problem in diagrammatic form. The ovals are items; the lines are links.
Here is part of the data for the bookseller problem in the same diagrammatic form.

Appendix 1 shows this schema in associative form and also in relational form, with the SQL and the relations that would be required to store an equivalent database.
7. CONCEPTUAL LAYER

The transition from things in the real world about which we want to record information to bytes on a disk in a database uses a modelling system to take us through three layers of abstraction: a conceptual layer, a logical layer and a physical layer. Each layer is less abstract and more concrete than its predecessor.

- The conceptual layer describes the conceptual building blocks that the modelling system uses to represent things in the real world, and sets out rules about how they may be used.

- The logical layer describes the logical building blocks which the database uses to store and access data, and how the conceptual building blocks are mapped on to them.

- The physical layer describes the physical building blocks which exist in the computer’s memory and are stored and retrieved in its disk storage, and how the logical building blocks are in turn mapped onto the physical ones.

The conceptual and the logical layers together make up the data model; the logical and the physical layers together make up the database management system. This chapter and the two that follow describe the three layers of the associative model.

**Entities and Associations**

Database management systems record the existence and properties of things in the real world. Application development methodologies and tools have used various different words, such as “entity”, “object” and “instance”, to express the idea of an
individual thing about which information is recorded. Each time
a word is used, it acquires a new set of semantic overtones that
are difficult to set aside: for example, it would be unthinkable
now to use the word “object” in the context of a database
management system without taking on board its object-oriented
connotations. For that reason, I have chosen to call those things
whose existence and properties are recorded by a database
simply “things”.

The associative model divides things into two sorts: entities
and associations.

- **Entities** are things that have discrete, independent existence.
  An entity’s existence does not depend on any other thing.
  Some types of things that would be represented by entities
  are people, cities, countries, books, vehicles, buildings,
  corporations and other legal entities.

- **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. Some types of things that would be represented
  by associations are employees, customers, contracts,
  marriages, journeys, production runs, and corporate
  headquarters. For example:

  - An employee is an association between a person and a
    legal entity.
  
  - A customer and a contract are associations between two
    people, a person and a legal entity or two legal entities.
  
  - A marriage is an association between two people.
  
  - A journey is an association between whatever is
    travelling – a vehicle or a person – and a route. A route is
    itself an association between an origin and a destination.
• A production run is typically an association between a product and a date/time, and a production facility.

• A corporate headquarters is an association between a corporation and a building or a location.

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.

By asserting that entities and associations are two fundamentally different types of real-world things, the associative model separates two ideas: on one hand, the idea of a thing in the real world that has discrete, independent existence, and on the other hand the idea of the various different ways in which such a thing interacts with other things. Each such interaction is a thing in its own right, about which we may want to record information.

• A person is an entity, whilst 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, whilst 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, whilst 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 real-world association is represented within an associative database as an association between two other things, each of which is itself an entity or an association. So customers and orders are represented by the following constructs:
To decide whether a thing is an entity or an association, ask yourself whether there is any other thing in the real world which, if it ceased to exist, would render the thing in question non-existent or meaningless. If so, the thing is an association; if not, it is an entity.

This test must always be applied in the present, not in the past or in the future. Obviously a person could not exist but for the two other people who were its parents, but this does not mean that people are associations. Nor is this type of dependence in any sense physical: one might say that a building depends on the ground that supports it, such that if the ground were removed the building would cease to exist. But the building would not cease to exist: it would simply be transformed to a different state – a heap of rubble – but would still exist in the sense that we are using in the context of the associative model. Similarly a person who has died, even a person whose mortal remains have long since ceased to exist, is still an entity.

The associative model distinguishes entities and association for a simple and fundamentally important reason: echoing our discussion in Chapter 5, 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, the long term.

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, whilst 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.

**Why Associations?**

For those of us brought up with the relational model, thinking in terms of entities and associations instead of just tables is a departure. Moreover the concept of associations as representing things in the real world is not essential to the associative model: one can still use an associative database to good effect and take advantage of its key benefits even if one models everything as entities. So why does the associative model advocate the use of associations in this way? It does so because it is more true to life: in other words, it is a better model of information in the real world. In Chapter 5, we discussed in general terms why a modelling system should always be as true to life as possible. Let us now consider specifics.

Most modelling systems represent a customer as an entity in its own right with independent existence (albeit with dependencies on other entities). However, in reality, “customer” is not an independent entity: it is a name we give to a role that one
legal entity plays with respect to another. Suppose Company A is setting out for the first time to develop its operational applications, the first of which is sales order processing. When it models legal entities for the first time, in the context of sales order processing, they appear in the role of customers, so Company A models them as single, independent entities. A customer entity has attributes of name, address, telephone number, buying contact’s name, credit limit and so on.

Having implemented sales order processing, Company A turns to its purchasing system. Now it needs to model legal entities for a second time, this time in the role of suppliers. It has already modelled them once, as customers, so can it reuse its customer entity? No, because each customer’s attributes include both those related to its existence as a legal entity, and those related to its role as a customer, and the latter would be inappropriate in its role as a supplier. So legal entities that are suppliers must be modelled separately, as supplier entities. This involves repeating work already done in modelling the attributes that relate to a supplier’s existence as a legal entity: name, address, telephone number and so on. When Company A wants to communicate with all of its customers and suppliers – perhaps following its change of address – two mail shot programs have to be developed and run, one for customers and one for suppliers.

Moreover, it is not unusual for one legal entity to operate in the roles of both customer and supplier with respect to another. Many opportunities are lost by modelling them separately. Suppose Company B is such a legal entity. When Company B changes its address, Company A has twice the work to do. Also Company A loses the opportunity readily to pool all of its information about Company B. Applications and users alike should be routinely aware of the big picture concerning Company B, such as for example when Company A owes Company B £5,000 in its role as supplier, whilst at the same time is owed £10,000 by Company B in its role as customer. When Company B’s two roles are modelled separately, it
requires more work from programmers and users alike to ensure that Company A’s interests are protected. A company for whom I built sales and purchase applications in 1980 estimated that the 5% of its business partners with whom it had both customer and supplier relationships consumed 50% of the time of its accounts department.

There is another aspect to this approach to modelling. Early in its life, when it was first developing its sales order processing system, Company A was a single legal entity, so the question of which legal entity owned a customer relationship did not arise: there was only one. Consequently it modelled its customers as individual entities with no capability to record that some legal entity other than Company A might own a customer relationship. As Company A grew, like most successful companies it soon established its first subsidiary company as a separate but affiliated legal entity. At this point its desire to retain a centralised account receivable function forced it to modify its system heavily to introduce this capability. The associative model would have prompted it to model customers as an association between two legal entities from the start.

Each of the approaches advocated here – separating the notion of a customer into its two aspects of a legal entity and its role as a customer, and modelling a customer as a relationship between two legal entities – can be implemented using the relational model via a modelling system that represents real-world associations as independent entities. But they are not natural features of the relational model, and both lead to a proliferation of tables that may be viewed as unnecessary complexity. Also there are no mechanisms within the relational model to guarantee the integrity of the essential associations that are inherent in this approach. Under the relational model, a sales order may survive the deletion of the customer who placed it, thus rendering it meaningless. By contrast, under the associative model, when a sales order is modelled as an association with a customer, deletion of the customer necessarily destroys its
orders, and conversely an order cannot exist without a customer. This is because links whose sources or targets cease to exist also themselves cease to exist.

Under the associative model, the designer’s decision on whether to model something in the real world as an entity or an association is theirs and theirs alone. The associative model advocates and rewards the use of associations, but does not mandate it.

Attributes as Associations

The pieces of information that a database records about a thing are called its attributes. In the real world, we describe things by associating them with other things. When we say the sky is blue we are describing the sky by associating it with blue. The two things that are connected are the source and target respectively. The connecting verb is the nature of the attribute.

Natures are usually verbs, sometimes abbreviated to prepositions. In everyday speech, we often mix together the words that represent targets and natures: we say Simon has two legs instead of Simon has number of legs Two. Also we often omit the nature altogether when it can be safely inferred from the targets: the sky is blue meaning the sky is coloured blue.

There are two different ways to represent an attribute in a database management system, depending on whether or not the target of the attribute is represented as an entity within the database:

- If the attribute’s target is represented as an entity, then the attribute is represented as a link between the source and the target: Order 123 was placed by Customer 456.

- If the attribute’s target is not represented as an entity within the database, then the attribute is represented by a value that has no identity and exists solely for the purpose of
expressing this single attribute: Customer 456 is called “Avis”. The value is part of the source’s representation in the database, and is local and private to it. Other attributes may target identical values, but the values are repeated every time, and no attempt is made to relate second and subsequent occurrences of the value in the database to the first. Such values are identical, but are not the same value.

(“Identical but not the same” is an important distinction: Consider three variables A, B and C, where A = 27; B = twenty-seven; C = the value expressed by A. If I change A, the value expressed by B doesn’t change, but the value expressed by C does. A, B and C all refer to identical values, but only A and C refer to the same value.)

Using the first approach to record an address, each city would be represented as an entity in its own right, so that every city is represented once and once only in the database, and the unique representation is referenced by every address or any other piece of data that includes the city. Using this approach with the relational model, each city would be represented by a tuple in a City relation.

Using the second approach, the city would simply be part of the text of the address, so that each city is potentially represented many times, and no attempt is made to relate one representation of a city to any other.

The associative model uses only the first approach. All attributes are represented as links between things within the database, and the target of every attribute is another thing that is represented within the database in its own right.

In the relational model, the source of an attribute is a tuple, the nature is a column heading of the relation containing the tuple, and the target is the value contained by the cell under the column heading in the tuple. In one sense, the relational model uses both of the approaches that we have described: when the
column is a foreign key, the attribute’s target exists as an entity – actually, a tuple in another relation – within the problem domain, and when the column is not a foreign key, the attribute’s target does not exist as an entity within the problem domain.

(I said in one sense: in another sense, the relational model does not deal explicitly in links at all. All attributes are represented by values in columns: some columns – the foreign keys – contain values that match the values in the corresponding primary key columns of a tuple in another relation, whilst other columns contain values that are local and private to the tuple in which they occur.)

The object model has no single approach to this issue. Most languages, with the exception of Smalltalk, use the concept of basic data types that are treated as values, not objects. In Smalltalk, everything, including scalar values and strings, is an object.

So in the associative model, attributes are represented as links between the entity or association whose attribute we are recording as the source, a verb to express the nature of the attribute, and an entity or association as the target. But we have already said that associations are links between two things, so this means in practice the attributes and associations are indistinguishable. So, in the associative model it is sufficient to say that things are described by means of their associations with other things, and attributes are no different from associations in general. This is good news, because at any time we may decide to describe an attribute by giving it attributes of its own. Under the relational model, this would mean restructuring the database, replacing a value by a foreign key and adding a new relation.

A significant component of many data models cannot simply vanish, so what has become of attributes, in the sense of non-foreign key values in the columns on a relation, in the associative model? The answer is that they are simply associations that are not themselves the source of any further
associations. In our example, Legal entity sells Book and (Legal entity sells Book) worth Points are both associations: the latter might be termed an attribute because it has no associations of its own. Having said this, let me reiterate that the notion of an attribute as distinct from an association plays no part in the implementation of the associative model: things are described solely by means of their associations with other things.

Modelling attributes as associations affords us an important opportunity that the relational model denies us – namely, the chance to record information about attributes. In its simplest form, a credit control application might say <Customer> has credit limit Value. This may suffice for a time, but as our credit control grows more sophisticated, it may become useful to know when a customer’s credit limit was increased or decreased to its current value. The associative model allows us to add attributes to our attribute, as in (<Customer> has credit limit Value) as of Date. Under the relational model, this process of replacing a column with a table, and the column’s domain with the foreign key of the table is one of the commonest and most disruptive of changes.

The Information Feature

The foregoing lets us assert the associative model’s equivalent of Codd’s Information Feature for the relational model, which says, in essence, that all information must be cast explicitly in terms of values in relations, and in no other way. The equivalent principle for the associative model is that all information must be cast explicitly in terms of associations, and in no other way.

Entity Types and Entities

There are two ways to describe a thing. We can list all of its attributes individually one by one: Michael Peters has two legs, two
arms, one head, two eyes and so on. Or, more efficiently, we can say that a thing is a member of a group of things that has a set of attributes in common, and that by virtue of its membership of the group, it acquires its own values for each of those attributes. So if we say Michael Peters is a Human being, then it follows that he has two legs, two arms and so on.

In the associative model, collections of similar entities are represented by entity types. An entity’s membership of an entity type is recorded by means of an entity type assertion, which is a link between the entity and the entity type, using the verb is a. Each entity is an instance of its entity type, and is said to instantiate the entity type. Entity types have some features in common with relations and domains from the relational model, and with classes and intrinsic types from the object model.

Each entity type has a number of association types. Each association type describes a particular association that each instance of the entity type may have. Thus the association type Person has date of birth Date means that every person will have a date of birth attribute. Association types are links that have the entity type itself as their source, and an entity type or association type as the target, with a verb that indicates the nature of the association type.

Identity

Entities are unique, individual things, tangible or intangible. Entities have identity: that is, they are capable of being identified unequivocally and distinguished from other, similar entities within the problem domain.

Identity is a subtle concept. In everyday speech we often refer to things by their types, knowing that our audience can fill in the blanks from their understanding of the context: “I’m going to the supermarket”; “Pass me the pen”; “Where are the kids?”; “Get into the car” are all examples of referring to things by their type. In the real world, identifying something
unequivocally is not always as easy as it sounds. To express “I’m going to the supermarket” using identities instead of types I would need to say something like: “I, Simon Guy Williams, British citizen with National Insurance number XY 12 34 56 Z, am going to the Tesco supermarket in Loudwater, Bucks, England”.

In the world of application development we routinely deal with abstractions and types, so we need to be particularly careful filling in the blanks. When we say “a person has a date of birth” we usually mean “every person has their own date of birth”. Here we are using the word person to refer to a type of entity, and thus to all entities of that type. When we say “I spoke to a person”, we mean “I spoke to an individual person”. Here we are using the word person to refer to an entity.

“Capable of being identified” doesn’t mean that an entity is necessarily already identified unequivocally, but simply that if we decide that we need to identify it we could do so. In other words, having identity is not the same thing as having an identifier. An identifier may be a key, a surrogate key, or a surrogate, that uniquely identifies an entity within the problem domain.

- A key is some combination of one or more of an entity’s existing properties that identifies it uniquely.

- A surrogate key is a new property created solely for the purpose of identifying an entity within the problem domain of the database, and its use for any other purpose is prohibited or strongly discouraged. A surrogate key is a key, and behaves in all respects in the same way as a key, being visible to the user, but its values are typically drawn from one simple domain.

- A surrogate, like a surrogate key, is created solely for the purpose of identifying an entity. Additionally it is assigned
The Associative Model of Data

by the system and is never visible to the user under any normal circumstances.

An entity may have any number of identifiers, including zero. Every entity has identity, but not all entities have an identifier – every grain of sand is an entity, but few of them have identifiers. Something that purports to have an identifier may not necessarily be unique: any implementation of the relational model that fails to expressly prohibit duplicate tuples lays itself open to this state of affairs. A thing that is not unique and does not have identity is not an entity but a type. The name “Michael Peters” on its own is not an identifier except within a very limited problem domain: there are probably several hundred or more Michael Peters is the UK alone; hence Michael Peters is a type.

In the associative model, every entity is assigned its own identifier in the form of a surrogate as soon as it is created. To reiterate, a surrogate exists solely for the purpose of identifying an entity to the database management system. It is assigned by the database management system, and is never seen by the user. Given that there is an entity for every scalar and every string, and that surrogates are never re-used, there needs to be a large supply of them in each database. There is no practical limit to the number available: in the current implementation there are \(2^{48}\) (more than 281 trillion) available in each chapter for each type of item, and the number of chapters in a database is limited only by the number of possible paths or URLs in the world.

Names

Surrogates are invisible, so database users visually identify entities by name and type. Every entity has a name, which is a string of unlimited length. Because the system identifies entities by surrogate key, it is indifferent to duplicate names, so users are able to specify how they would like duplicate names to be
treated: usually duplicate names within a single database are not a good idea and users will ask the system to prevent this happening by prohibiting the creation of duplicate names.

However, the associative model allows separate databases to be freely merged or viewed as one, and in this situation duplicate names are almost inevitable, and moreover not inappropriate. When duplicates occur, the user will inspect each entity and its immediate associations to decide which real-world thing it represents and, depending on the circumstances, either choose the appropriate one, or alter the name of one to remove the ambiguity.

Because the system does not rely on names for identity, names can be freely altered at any time to meet the purpose the user requires. In the relational model, if the value in a foreign key is altered, the relationship between the entity represented by the tuple containing the foreign key and the entity represented by the tuple whose primary key was previously stored in the foreign key is no longer recorded.

**Association Types and Associations**

Collections of associations that have associations in common are represented by association types, and an association is an instance of its association type. Association types have the following properties:

- **Name**: appears as the verb in associations that instantiate this association type.

- **Source type**: the source of associations which instantiate this association type must be of this type.

- **Target type**: the target of associations which instantiate this association type must be of this type.
• **Cardinality**: determines how many instances of this association type may share the same source.

• **Inverse cardinality**: determines how many instances of this association type may share the same target.

• **Sequenced or sorted**: determines whether multiple instances of this association type that share the same source are presented in the natural order of their targets (“sorted”), or in an order determined by the user (“sequenced”).

• **Default target**: determines the target that an association is deemed to have by default when the association type is not actually instantiated.

An association type also determines which associations, if any, its instances may have, by means of its own association types. Again, paralleling entity types, these are links that have the association type itself as their source, and an entity or association type as the target, with a verb that indicates the nature of the association. This gives us:

```
Person customer of Legal entity
  ... placed an order on Date
  ... for Book
  ... in quantity Quantity
```

**Inverse Associations**

In the real world, when A has a relationship with B, B has some type of relationship with A. Sometimes this inverse relationship is significant; sometimes it is trivial. If Fred is married to Mary, it is probably significant for Mary that she is married to Fred; on the other hand, that the earth orbits the sun is significant for the earth, but that the sun is orbited by the earth is of very little significance for the sun.
In the associative model, every association type has an inverse, either explicit or inferred, which expresses the opposite voice (active or passive) to the verb itself. Where it will add more meaning, a verb to be used for instances of an association type’s inverse associations may also be specified as part of the association type definition:

```
Person customer of Legal entity
... inverse verb supplier of
```

When the inverse is not specified, is inferred according to the following rules:

- If the verb begins with “has”, the “has” will be removed and the suffix “of” will be added: “has customer” becomes “customer of”.

- If the verb ends with “of”, the “of” will be removed and the prefix “has” will be added: “customer of” becomes “has customer”.

- If the verb neither begins with “has” nor ends with “of”, the suffix “of” will be added: “order date” becomes “order date of”.

- If the verb both begins with “has” and ends with “of”, the prefix “inverse of” will be added: “has order date of” becomes “inverse of has order date of”.

Although inverse associations do not physically exist as links, they are always included in queries unless the metadata indicates as part of the association type definition that they are not to be.

**Cardinality**

We mentioned an association type’s cardinality and inverse cardinality above. The cardinality of an association type is one
of four values: “Zero or one”; “One”; “One or more”; “Any number”. Any association type for which cardinality is not explicitly specified is assumed to have cardinality One. The effect of the various cardinalities on the association type Person orders Book is as follows:

- **Zero or one**: a person may order no books or one book.
- **One**: a person orders one book and one only.
- **One or more**: a person must order at least one book.
- **Any number**: a person may order any number of books from zero upwards.

An association type may also have an inverse cardinality, which in this example would say how people may order a book. Cardinality itself says nothing about its inverse: the fact that a person must order at least one book says nothing about how many people may order a single book. If the inverse cardinality is to be checked, it is recorded as part of the association type’s definition.

- **Zero or one**: a book may be ordered by no more than one person
- **One**: a book must be ordered by one person and one only.
- **One or more**: a book must be ordered by at least one person.
- **Any number**: a book may be ordered by any number of people from zero upwards

Inverse cardinalities of “one” and “one or more” are not as useful as the other two, because, given that we have chosen to assert that the association type points actively from person to book, it is reasonable to suppose that books are passive in the
association, and thus do not know or care how many people order them. Moreover, if a type is the target of an association type whose inverse cardinality is “one” or “one or more”, we reach deadlock when we try to create a new instance: it can’t exist unless something points to it, and nothing can point to it until it exists.

**Default targets**

Most association types have different targets each time they are instantiated – **Person born on Date** is a good example – but sometimes every instance of an association type has the same target most or all of the time. In this case, a default target may be recorded for the association type as part of the metadata. When an association that instantiates an association type with a default is about to be created, if the association’s target is the same as the default, the association is not created, but is inferred with the default target in all queries. For example:

\[(\text{Human being has number of legs Zero, one or two}) \text{ default Two} \]

implies that every instance of entity type human being has two legs.

**Inferred Associations**

As we have seen, there are two cases where the existence of associations that do not physically exist is inferred by the system: inverse associations and associations whose target is a default. Several points arise:

- During queries and information retrieval within the system, inferred associations behave like real ones in every respect.
• When data is exported from a database via ODBC, JDBC or other middleware component, inverse associations are not included.

• The inverse association of an inferred default association is also inferred.

**Instance association types**

The associative model permits an association type that is specific to a single entity or association. This is modelled as an association type whose source is an instance, not a type. Associations that instantiate this association type are created and maintained in the normal way: the single additional rule is the source of such associations must be the entity or association that is the source of the association type.

**Subtypes and Supertypes**

A entity or association type may be the subtype or supertype of another type. An instance of a subtype has associations that instantiate the association types of its type, and also those of its type’s supertypes and their supertypes. This is commonly interpreted as inheritance. So:

```
Tangible object weight Kilos
Car subtype of Tangible object
Car number of wheels Four
Car number of seats Between one and five
V123ABC is a Car
```

... leads us to:

```
V123ABC weight 1,759 kilos
V123ABC number of wheels Four
V123ABC number of seats Five
```
A type may have multiple supertypes, so we could add to the above:

Vehicle maximum speed *Miles per hour*
Car subtype of Vehicle
V123ABC maximum speed *150 mph*

Some entity and association types are abstract: that is, they are not intended to be instantiated, and exist solely to add to the set of association types that instances of their sub-types may instantiate. For example, it might be appropriate to decide that *Tangible object* was an abstract entity type as long as we are sure that our application will never need to deal with entities whose sole properties are those of a tangible object.

When the target of an association type is a supertype, members of the supertype itself and its subtypes are candidates as targets of the associations that instantiate the association type. Similarly, when we ask an associative database to show us all instances of a type that has sub-types, we would expect to see all instances of the type itself (if it has any) together with all instances of each of its sub-types.

Although we think most readily in terms of entity subtypes and supertypes, the sub- and supertype capabilities apply equally to both entity types and association types. Here is an example of an association subtype and supertype:

*(Animal has number of limbs Integer) supertype of (Human being has number of arms Small integer)*

Entity supertypes and subtypes need only be semantically sensible, but for association supertypes and subtypes there are some formal rules to be observed. For one association type to be a subtype of another:

- The subtype’s source must be the same as or a subtype of the supertype’s source
• The subtype’s target must be the same as or a subtype of the supertype’s target

So our example above requires that:

- **Human being** is a subtype of **Animal**
- **Small integer** is a subtype of **Integer**

Sub- and super-typing has two main uses. First is the classic “specialisation/generalisation”, or inheritance, mechanism, that permits the definition of a class to re-use the definition of a more abstract class. This is best exemplified by the taxonomy of living and extinct organisms. For example, we re-use the definition of vertebrate to define Homo sapiens.

Secondly, abstract supertypes are often a useful way to group together otherwise heterogeneous things that form a group for some special purpose. For example, contractual parties may be corporate entities or individuals:

- **Contract party** **Contractual party**
- **Person** subtype of **Contractual party**
- **Company** subtype of **Contractual party**

may be a more elegant way to implement:

- **Contract party** **Person**
- **Contract party** **Company**

**Subsets and Supersets**

As well as entity subtypes and supertypes, the associative model makes use of inferred subsets. A subset is a type that behaves as a subtype, except that it may not be the target of an entity type assertion: in other words, its membership is inferred, not explicitly declared. For example: **Good customer** subset of **Customer**. Having said that **Good customer** is a subset, we would no longer be allowed to assert that **XYZ** is a **Good customer**. As a
corollary, if an entity type has explicitly declared members, it cannot be a subset. A subset is populated with the result set of a query:

**Good customer** subset of **Customer**
... populated by **Good customers query**

The difference between subsets and subtypes is that membership of subsets is inferred, and therefore transient, whilst membership of subtypes is asserted and permanent. A subset is automatically a subtype of its superset. A subset may have subsets, but a subset may not have more than one superset.

Subsets are a useful mechanism to keep track of the various states that an entity or association may pass through. For example, a credit control system may track customers as new, established, premium, watch or stop according to their membership of inferred subsets that are based on queries run over their historic credit and buying patterns. Customers will automatically move into and out of subsets as their history develops.

Subsets are also useful as a validation mechanism. If our sales order processing system only accepts orders for Items in stock, we might define a subset of the entity type **Item** called **Item in stock**. We may then use **Item in stock** as the target of an order association: `<Order>` placed for **Item in stock**. Only items that are members of the **Item in stock** subset are then eligible to be entered on an order.

### Scalars, Strings and References

As well as entities and associations in the sense that we have been discussing them, there are three other types of thing that may be stored: scalar values, strings and references. Scalar values (or scalars for short) express magnitude, such as numbers, quantities, monetary values, amounts, prices, ratios, dates, time
intervals and so on. Strings are strings of characters, such as names, descriptions, codes and short, unstructured pieces of text. References are pointers to things that live outside the database but are referenced from within it. These include multimedia files, web site URLs, email addresses and so on. Together, these three types of thing represent what the relational model calls “values”. They are all atomic from the database’s point of view.

In the associative model, scalars, strings and references are treated as entities. Thus each individual scalar, string or reference is stored once and once only, and there never needs to be more than one occurrence of “1”, or “17-Jan-1998”, or “$100”, or “15%”, or “London” in a database. Notice that I said “there never needs to be”, not “there is never”. In practice, because the associative model allows separate databases to be freely merged or viewed as one unified database, often there will be more than one occurrence of a particular scalar, string or reference in a database. However, because they have no properties other than their name and type, the database management software ensures that there are no adverse consequences of representing a scalar, string or reference more than once in a database.

Scalars, strings and references may not be the source of any association types (because they are atomic from the database’s perspective) and are associated with a datatype. Datatypes in the associative model are “soft”; that is, they comprise an open-ended set which may be added to by vendors and users alike. The implementation of a datatype must deal with the following:

- Transformation of values from a format suitable for physical storage to a visual representation (including, in the case of references, appropriate rendering methods).

- Parsing and transformation of values from a visual representation as a series of key-strokes to a format suitable for physical storage.
• Precedence rules for sorting values into their “natural” sequence.

• Implementation of the basic arithmetic and comparison operators (add, subtract, multiply, divide, less than, greater than, equal, not equal) between two instances of itself, and between one instance of itself and an instance of any other datatype.

Creating Associative Schemas

Creating any database schema entails two sorts of activity: analysis and design. Analysis involves recognising and understanding the elements that make up the problem domain, and how they relate to one another. Design involves deciding which of the different types of database building blocks should be used to represent each element. Analysis and design may be undertaken one after the other, as in the older, more formal waterfall style of development, or may be undertaken together, as in the newer, less formal RAD style of development. If the analysis is accurate and thorough, the design process is usually straightforward. If the analysis is faulty or superficial, the design process will be difficult and will not yield a good result.

The relational model and the associative model both address the same types of problem domain, and the tasks performed during analysis are broadly the same in each case. The differences between the relational model and the associative model occur in the design phase, because the associative model offers a different set of building blocks from which to construct a schema. In other words, once the problem domain has been analysed and understood, the two models offer alternative ways to implement an application based on it. Someone skilled in relational schema design will find that their analysis skills are equally relevant to the associative model, so they are more than
half way to becoming skilled in designing associative schemas. The transition to associative schema design primarily involves becoming familiar with the building blocks provided by the associative model, and learning how best to use them.

Comparing the building blocks of the associative model to those of the relational model, we can make the following general observations (to which there will always be exceptions):

- An entity type that is the source of one or more association types is equivalent to a table that has no foreign keys as primary keys.

- An entity type that is the source of no association types is equivalent to a domain.

- An association type is equivalent to a table that has one or more foreign keys as primary keys.

- An association type’s verb is equivalent to a column heading in a relation.
8. LOGICAL LAYER

Now we shall look more closely at the logical building blocks of the associative model: items, links, chapters and databases. As we said at the start of Chapter 6, under the associative model, a database comprises two data structures:

- A set of items, each of which comprises, amongst other things, a unique identifier, a name and a type.

- A set of links, each of which comprises, amongst other things, 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 of the three things identified by the source, verb and target may each be either a link or an item.

The third type of building block is the container for items and links, which is called a chapter. A chapter contains a subset of the items and a subset of the links in a database, so each database comprises one or more chapters. Each chapter also contains a list of the chapters that contain the items and links whose identifiers occur as the source, verb or predicate of one or more of the chapter’s own links. Such chapters are identified within a chapter by an identifier local to the chapter.

**Items**

The properties of an item are as follows:

**Identifier**

An item’s identifier identifies it uniquely within the scope of its chapter and its type. The identifier is a surrogate, that is
automatically assigned by the system to each new item as it is created and subsequently never changed. It exists solely to allow the system to identify and distinguish items, and is never seen by any developer or end-user. Identifiers of items that are removed from the database are never re-used.

The number of unique identifiers required by an associative database is of the same order as the number of cells (one column of a tuple) in a relational database. (The number is higher in that identifiers are never re-used, lower in that scalar values are represented as entities and thus occur once only, and lower in that no extra columns are required for foreign keys.) As a reality check, consider the hypothetical relational database of about 500 relations whose size is analysed in the following table. The columns mean that there are 8 relations of 10 million tuples with an average of 20 columns each, 16 relations of one million tuples with an average 20 columns each, and so on. As the table shows, the number of cells in such a database is about 2 billion.

<table>
<thead>
<tr>
<th>Number of relations</th>
<th>Number of tuples per relation</th>
<th>Average number of columns</th>
<th>Total number of columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>10,000,000</td>
<td>20</td>
<td>1,600,000,000</td>
</tr>
<tr>
<td>16</td>
<td>1,000,000</td>
<td>20</td>
<td>320,000,000</td>
</tr>
<tr>
<td>32</td>
<td>100,000</td>
<td>20</td>
<td>64,000,000</td>
</tr>
<tr>
<td>64</td>
<td>10,000</td>
<td>20</td>
<td>12,800,000</td>
</tr>
<tr>
<td>128</td>
<td>1,000</td>
<td>20</td>
<td>2,560,000</td>
</tr>
<tr>
<td>256</td>
<td>100</td>
<td>20</td>
<td>512,000</td>
</tr>
<tr>
<td>504</td>
<td></td>
<td></td>
<td>1,999,872,000</td>
</tr>
</tbody>
</table>

An associative database with an identifier space of $2^{48}$ (about 2.8 $\times 10^{14}$) would have sufficient identifiers to contain a snapshot of this hypothetical database 140,000 times over. A relational database of 500 tables with 8 tables of 10 million rows is in the top quartile of databases in use today. Nevertheless, in an implementation of the associative model the identifier space should always be as large as possible, and not less than $2^{48}$. 
Name

An item’s name is the piece of information by which the user visually identifies the item within its type. The name is not an attribute of the item in the usual sense. It is not part of the data that we store about an entity, but rather it is data that we use to help us to store data: in other words, a handle for the stored data. An item’s name bears the same relationship to the data that we store about the item as the name of a file containing a Word document bears to the content of the document itself.

This also makes sense in the context of applications. We frequently use names for the entities with which we work that are not needed as part of the entities’ stored data. Take IBM’s customer centre on the south bank of the River Thames in London. Universally referred to as “IBM South Bank”, its legal entity name is “IBM United Kingdom Ltd”, and its postal address is “76/78 Upper Ground, London SE1 9PZ”.

A name is a character string of any length. Ideally, any and every character that has an external representation should be valid within a name, in any combination, although this objective may be compromised by the practicalities of implementation.

Names of all types of items except scalar values are purely descriptive. An item’s name is not part of the data that the database records about an item and is not subject to the same rules and mechanisms. The item name should not be used or relied on as part of an entity’s data. The appropriate way to represent the name of the entity Michael Peters would be:

Michael Peters forename Michael
Michael Peters second name John
Michael Peters family name Peters

... and if the name of the item “Michael Peters” were changed to “Mary Peters”, until updated the database would continue to say:

Mary Peters forename Michael
Mary Peters second name John
Mary Peters family name Peters
This is also relevant to enterprises, the name of whose entity would typically be that by which the enterprise is commonly known. So we would have, for example:

**IBM legal entity name IBM United Kingdom Ltd**

Generally, an item’s name is unique within its type and its database. However, there is no absolute requirement for names to be unique, and duplicate names within a type are handled by the database in the normal course of events.

With one exception, the character string of an item’s name is never altered. The general case is that, when the user needs to change the name of an item, and provided they are authorised to do so, the system effects the change as follows:

1. A new, archived item of the same type and name as the subject item is created.
2. A new, archived link is created between the subject item and the new item using the verb “name changed from”.
3. The name of the subject item is changed to the new value entered by the user.

The sole exception is that the character string of an item’s name may be altered by the user who created the item within the scope of the transaction that creates the item. (Transactions and their scope will be discussed later.) This is to avoid the creation of spurious items through mis-typing.

The names of scalar values are their values. Such names have an internal and an external representation. The internal representation is what is stored on disk and in memory, and is in an implementation-specific format; the external representation is what is seen – that is, which data is displayed or printed. The representation of a scalar value is converted from internal to external and vice versa according to rules provided by its datatype. This is similar to the way in which spreadsheets store and present values.
Item Type

Items represent the following types of things:

- **Entity type** ("E-type"): An entity type is an abstraction that represents a collection of real-world entities which have properties in common. Entity types have features in common with classes, domains and relations.

  Entity types may be real or abstract. Abstract entity types may not be instantiated by entities, but exist as templates from which more concrete entity types may inherit properties.

- **Association type** ("A-type"): An association type is an abstraction that represents a collection of similar associations between things in the real world. Association types have features in common with classes and relations.

  Association types may be real or abstract. Abstract association types may not be instantiated by associations, but exist as templates from which more concrete association types may inherit properties.

  Association types are items whose properties are associated with it by links. The association type item’s name appears as the verb in associations that instantiate it. There are three categories of association type:

- **Regular association types** are templates for associations that may be instantiated (as opposed to inferred) for each instance of the association type’s source. The source and target of a regular association type are types.

- **Irregular association types** are templates for associations that may be instantiated only on the instance
that is the source of the irregular association type itself. The source of an irregular association type is an instance and its target is a type.

- **Type association types** are templates for associations that are inferred (not instantiated) for each instance of the association type’s source. The source of a type association type is a type and its target is an instance.

When the distinction between the three sorts of association type is important the full name is used. When the term association type is used without qualification, the distinction is immaterial.

- **Entity**: An entity represents a discrete, independent thing in the real world. whose existence and properties are recorded in a database.

- **Verb**: A verb expresses the nature of a link. There are two sorts of verb:
  - **System verbs**, whose meaning has particular significance to the system; and
  - **User verbs**, that occur as the verbs and inverse verbs of association types and associations, whose meaning is purely expressive and has no significance to the system.

- **Query**: A query is a prescription to selectively retrieve and operate on information from the database.

- **Transaction**: A transaction is a single database transaction that has the properties of atomicity, consistency, isolation and durability.
• **Aspect**: An aspect is a view of the database including certain types and ranges of entities and associations and excluding others.

• **Literal**: A literal is a character string expressing a rule about meta-data that is significant only to the database management system itself.

### Links

The properties of a link are as follows:

#### Identifier

A link’s identifier identifies it uniquely within the scope of its chapter and its type. The identifier is a surrogate, that is automatically assigned by the system to each new link as it is created and subsequently never changed. It exists solely to allow the system to identify and distinguish links, and is never seen by any developer or end-user. Identifiers of links that are removed from the database are never re-used.

The properties of the link identifier are the same as those of the item identifier except that they are assigned from a different range.

#### Source, Verb and Target

The source, verb and target of a link are each a reference to an item or a link. All such references comprise:

• **Chapter identifier**: If the referenced item or link is not contained within the same chapter as this link (the “home chapter”), the chapter identifier is a reference to an entry in the home chapter’s chapter list. This list in turn contains the
path, URL or other address where the referenced chapter may be found. If the referenced item or link is contained within the home chapter, the chapter identifier is null.

- **Item or link identifier** of the item or link referenced.

Certain types of link have no target. In this case, both the chapter identifier and the item or link identifier are null.

**Originating transaction**

Each link carries a reference (chapter identifier and item identifier) to the transaction that created it. Transactions are represented within the database as items. There are two possible cases:

- If the referenced transaction exists as an item within the database, then the link has been created by a completed transaction and the data that it represents is part of the database.

- If the referenced transaction does not exist as an item within the database, the link has been created by a transaction that is still in progress or has aborted, and the data that it represents is not part of the record. Such links are always ignored.

**Link Types**

The type of a link is not recorded explicitly on the link: it can invariably be determined from its verb. Links represent the following sorts of things:

- **Association**: An association is a link that represents an association between two things in the real world. Each association instantiates an association type. Its source is an entity or an association of the same type as the source of the
instantiated association type; and its target is an entity or association of the type specified as the target of the instantiated association. There are three categories of association type:

- **Regular associations** instantiate regular association types: that is, those whose source is a type.

- **Irregular associations** instantiate irregular association types: that is those whose source is an instance. Its source is the same as that of the association type.

- **Type associations** instantiate type association types: that is, those regular association types whose target is an instance. Type associations are not persistent: their existence is inferred for each instance of the source type.

When the distinction between the three sorts of association is important the full name is used. When the term association is used without qualification, the distinction is immaterial.

- **General type properties**: Association types and entity types have the following properties in common, which are recorded by links of which the type itself is the source.

- **Supertype assertion**: A supertype assertion records that the source entity type or association type has a supertype, which is the target entity or association type. The verb is the system verb “has supertype”; and the target is a type of the same sort – entity or association – as the target.

If the source and target types concerned are association types, it is a requirement that (a) the source type of the source association type is identical to or a subtype of the source type of the target association type, and that (b) the
target type of the source association type is identical to or a subtype of the target type of the target association type.

- **Subset assertion**: A subset assertion records that the source entity type or association type has a subset which is the target entity or association type. The verb is the system verb “has subset”; and the target is a type of the same sort – entity or association – as the target.

  A type may have any number of subsets, including zero. A type may not have more than one superset. A subset may have subsets of its own.

- **Subset query link**: A subset query link determines the query that is used to test for membership of a subset. Its source is the subset assertion; the verb is the system verb “membership query”; and its target is a query. The type of the query result must be the same as or a superset of the type of the subset.

- **Abstraction flag**: An abstraction flag link determines whether the type is abstract, which means that it may not be instantiated. The verb is the system verb “abstract”; and its target is one of the literals “Yes” or “No”.

- **Association type properties**: Association types are represented by items. They have properties not shared by entity types, which are recorded by the following types of link. In each case the association type is the source of the link.

  - **Source type or instance**: The source type or instance of an association type is associated with it by means of a link whose verb is the system verb “source”; and whose target is a type (for a regular association type) or an instance (for an irregular association type).
• **Target type or instance**: The target type or instance of an association type is associated with it by means of a link whose verb is the system verb “target”; and whose target is a type (for a regular association type) or an instance (for a type association type).

• **Cardinality**: The cardinality of an association type is represented by a literal, and is associated with the association type via a link whose verb is the system verb “cardinality”; and whose target is one of the literals “one”, “zero or one”, “one or more” or “any number”.

• **Inverse cardinality**: The inverse cardinality of an association type is represented by a literal, and is associated with the association type via a link whose verb is the system verb “inverse cardinality”; and whose target is one of the literals “one”, “zero or one”, “one or more” or “any number”.

• **Inverse verb**: The inverse verb of an association type is represented by a literal, and is associated with the association type via a link whose verb is the system verb “inverse verb”; and whose target is a user verb.

• **Dependency**: A dependency link asserts that the existence of a particular association is either prohibited by or required by the existence of another association with the same source. The assertion is framed in terms of the association types that would be instantiated by the two associations. The verb is one of the system verbs “prohibited by” or “required by”, and the target is an association type. Both source and target association types must have the same source.
- **Entity properties**: Entities are represented by items. They have the properties which are recorded by the following types of link. In each case the association type is the source of the link.

- **Type assertion**: A type assertion records that a particular entity instantiates a particular entity type. The source is an entity; the verb is the system verb “is a”; and the target is an entity type. An entity must have one and only one type assertion.

- **Query properties**: Queries are represented by items. Their properties are implementation specific.

- **Utility links**: The following types of link may be found in conjunction with a variety of other types of link.

- **Equivalence assertion**: An equivalence assertion records the fact that something is represented twice in the database. The source may be an entity, an entity type, an association, an association type or a user verb; the verb is the system verb “equivalent to”; and the target is an item or a link of the same sort as the source.

When data is displayed or queried, all references to the source are replaced by references to the target: in other words, the target appears as the source, verb or target of links where it appears as such in its own right, and links where the equivalence link is referenced as the source, verb or target respectively. An item or link that is the source of an equivalence is not normally displayed or seen by a query. A special mode for display or query is used to reveal such items and links.
• **Stop link**: A stop link records the fact that the source is logically deleted. The source may be a link or an item; the verb is the system verb “is deleted”; the link has no target. When data is displayed or queried, the source and the stop link itself are ignored.

• **Sequence links**: Sequence links determine the sequence in which a related group of links is displayed. There are two sorts of sequence link:

  The first type begins each sequence. Its source is the first item or link in the sequence, or the parent of the sequence of items or links; the verb is the system verb “starts with”; and the target is the next item or link in the sequence.

  The second type records each subsequent step in the sequence. Its source is the previous sequence link; the verb is the system verb “followed by”; and the target is the next item or link in the sequence.

  Sequence links are mainly used to sequence the association types that form the predicates of a particular entity type or association type.

**Chapters and Profiles**

A database is stored in any number of files called **chapters**, each containing a subset of the items and links in the database. Every chapter also includes its own list of the chapters that contain the items and links whose identifiers occur as the source, verb or target of one or more of the chapter’s own links. These foreign chapters are identified within a chapter by an identifier local to the chapter, and the list includes the path name or URL that
allows the chapters to be located across a network or via the Internet.

Returning to the simple example which we introduced in Chapter 6, suppose now that the items and links are stored in two chapters: Chapter A, which is stored in \SRV1\DIR\FILE1.CHP, and Chapter B, which is stored in \SRV2\DIR\FILE2.CHP. The identifiers that form the source, verb and target of each link now also include a local identifier for the chapter where the item or link is located.

In Chapter A we have:

<table>
<thead>
<tr>
<th>Items</th>
<th>Identifier</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>787</td>
<td>Flight BA1234</td>
</tr>
<tr>
<td></td>
<td>332</td>
<td>12-Aug-99</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>arrived at</td>
</tr>
<tr>
<td></td>
<td>019</td>
<td>At</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Links</th>
<th>Identifier</th>
<th>Source</th>
<th>Verb</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>784</td>
<td>0/787</td>
<td>0/132</td>
<td>1/008</td>
</tr>
<tr>
<td></td>
<td>053</td>
<td>0/784</td>
<td>1/767</td>
<td>0/332</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapters</th>
<th>Local identifier</th>
<th>Chapter name</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>Chapter A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Chapter B</td>
<td>\SRV2\DIR\FILE2.CHP</td>
</tr>
</tbody>
</table>

In Chapter B:

<table>
<thead>
<tr>
<th>Items</th>
<th>Identifier</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>008</td>
<td>London Heathrow</td>
</tr>
<tr>
<td></td>
<td>488</td>
<td>10:25am</td>
</tr>
<tr>
<td></td>
<td>767</td>
<td>On</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Links</th>
<th>Identifier</th>
<th>Source</th>
<th>Verb</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>664</td>
<td>1/053</td>
<td>1/019</td>
<td>0/488</td>
</tr>
</tbody>
</table>
A database may comprise any number of chapters, and each chapter may be located anywhere on a network or on the Internet, provided only that it can access and be accessed by the other chapters.

The chapters in a database form a network of peers. From the standpoint of the database management system, individual chapters in a database have no special significance, although they may do so from the users’ point of view. The three types of data contained in chapters are metadata, data and transactions.

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.

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 U.S. might use three chapters for its schema:

<table>
<thead>
<tr>
<th>Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local identifier</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local identifier</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Chapter A: **Person** first name **Name**  
**Person** family name **Name**  
*Seen by all users*

Chapter B: **Person** nat’l insurance no **NI number**  
*Seen by UK users*

Chapter C: **Person** social security no **SS number**  
*Seen by U.S. users*

The profile mechanism depends on two key features of the associative model:

- 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.

**Updates**

Under the relational model, transactions update databases by creating, deleting and changing tuples in relations. (Terminology varies: I use the term “change” for the act of reading a piece of data, changing one or more of its values and re-writing it, and the word “update” to mean any alteration of a database’s state, including create, delete and change.) By contrast, under the
associative model, in the normal course of events data in a database is never physically deleted or changed. The process in the relational model whereby values in an existing tuple are physically altered to different values has no equivalent in the associative model. Instead, changes in an associative database are effected by logically deleting the appropriate links and adding new ones. A link is logically deleted by the addition of another link, called a stop link, which has the deleted link as its source. Thus, data is created, deleted and changed in an associative database by a single mechanism: the addition of new links. For example, in the bookseller problem, we had:

**Amazon sells Dr No**
... worth **75 points**

If Amazon now decides that Dr No is worth 100 points, the link with 75 points as its target is logically deleted by the addition of a new link, and a new link with 100 points as its target is added.

**Amazon sells Dr No**
... worth **75 points**
... deleted by **Transaction 97756392**
... worth **100 points**

Items are deleted, and their names changed, in a similar manner:

**Amazon deleted by Transaction 97756392**

**Peking renamed to Beijing**
... renamed by **Transaction 97756392**

Associations are frequently the source of other associations. When it is necessary to change an association’s target, instead of logically deleting it and creating a new one, which would necessitate recreating all the subordinate associations, it may be preferable to change its target:

**Amazon sells Dr No**
... target changed to **Doctor No**
... target changed by **Transaction 97756392**
... worth **75 points**
In implementation, the workings of this process are not visible to query functions and the database application program interface, to which the create, change and delete functions appear to operate as though data was being physically altered or deleted. The links that implement deletion, renaming and new targets are intercepted, interpreted and filtered out as soon after physical retrieval as possible by the database management system.

**Transactions**

A database reflects the state of some part of the real world. When something in the real world changes, the database must change too to ensure that it still accurately reflects the real world. The thing that causes the database to change in response to a change in the real world is called a transaction.

Transactions vary in length. A short transaction might correct a mis-spelling in an address, which would require an alteration to a single property of a single thing. A transaction of medium length might add a sales order for twenty different product lines to the database, which would require the addition of dozens of new things to the database, each with its own properties. A long transaction might add an enterprise’s complete line-item budget for a new fiscal year, which would require the addition to the database of hundreds or thousands of new things, each with its own properties, and might require the addition of some entirely new types of things. So a single transaction may initiate anywhere between one and several thousand updates to the database.

Regardless of how many updates it initiates, every transaction must be atomic, consistent, isolated and durable. Atomicity and durability are preserved by database recovery mechanisms, which handle various types of hardware, software and other system failures. Isolation is preserved by database concurrency mechanisms.
Database transaction processing, concurrency and recovery are all major subjects in their own right. The fundamental principles of each types of mechanism are relevant to all database architectures, including the associative model, although much of the literature naturally focuses on their application to the relational model. My treatment of these topics here is entirely superficial. Most introductions to database concepts, such as Elmasri and Navathe [2] and Date [3] will provide a good overview. For more depth, Gray and Reuter’s “Transaction Processing: Concepts and Techniques” [41], and Kumar and Hsu’s “Recovery Mechanisms in Database Systems” [42] are excellent.

The main distinction that arises with respect to the associative model is its “add only” update philosophy, which has implications for atomicity and isolation that are discussed below. Under the associative model, all updates, including changes and deletions, are accomplished by adding items and links to the database. Each transaction is also itself represented by an item, which has links to other items and links to record its own attributes. The attributes recorded may vary from one implementation to another, but would typically include who originated the transaction, and the date and time at which the transaction was committed.

In Lazy Software’s implementation, the atomic action that commits the transaction to the database is the addition of the link between the item representing the transaction and the item representing the timestamp that records the date and time at which the transaction as committed. We call this the “commit link”, and it is the last link in the following example:

```
XYZ Ltd address 125 Canary Wharf, London E6 5TT
... deleted by Transaction 97756392
XYZ Ltd address Montague Court, London EC3 4RR
... created by Transaction 97756392
Transaction 97756392 committed by Simon Williams
Transaction 97756392 committed at 2002/03/21 15:57:01.51
```
Our implementation uses a visibility layer which operates close to the physical storage medium. The job of the visibility layer is to filter out all items and links that are not currently logically visible for any reason. The visibility layer applies, amongst others, the following rules:

- Any transaction item whose commit link is not visible is itself not visible;
- Any item or link created by a transaction that is not visible is itself not visible;
- Any item or link logically deleted by a stop link created by a transaction that is not visible is itself visible.

Thus, during a transaction, the items and links that the transaction creates, including stop links, are added to the database but remain invisible because the transaction itself is invisible. At the commit point, the commit link is added and the transaction is thereafter visible. If the transaction fails to commit for any reason, including system failure, the commit link will not be added and the transaction is thus never committed. Items and links created by uncommitted transactions remain in the database, but are filtered out by the visibility layer, and may be physically deleted at any time by housekeeping routines.

**Atomicity**

Atomicity means that a transaction must be “all-or-nothing”: either the entire transaction is effective, or the transaction has no effect. If the transaction succeeds, then its entire effect is reflected in the database. If a transaction fails, then it has no effect on the database.

The associative model preserves atomicity through the commit mechanism described above. A transaction is rendered visible by the atomic action of adding its commit link to the
database. Once this has happened, the entire transaction is visible. If it fails to happen, none of the transaction is visible.

**Consistency**

Consistency says that a transaction must not effect changes that cause any of the rules and integrity constraints expressed about data in the database to be violated: it must leave the database in a consistent state.

**Isolation**

Isolation requires that a transaction must execute independently of any other transactions that may be executing at the same time (“concurrently”). It must appear to each transaction that every other transaction happened either before or after itself. The effect on the database of executing a number of transactions concurrently must be the same as if they were executed one after the other (“serially”).

In the associative model, the commit mechanism described above also serves to preserve isolation. From the standpoint of the visibility layer, every transaction occurs at the precise instant recorded by its commit link, and a transaction is rendered visible by the atomic action of adding its commit link to the database. Thus no transaction that is in progress and has not yet committed may ever see any part of any other transaction that may also be in progress.

**Durability**

Durability requires that, once a transaction has succeeded, the changes that it has made to the state of the database must be permanent. Once atomicity and consistency have been assured, the principal mechanism to ensure durability is the ability to make and restore backups of the database.
Recovery

Recovery is the process of returning the database to a consistent, correct state after the failure of some sort. Failures are of three types:

- **Transaction failure**: A single transaction fails to complete, typically due to an error in a program.

- **System failure**: A system failure is one that affects all transactions in progress at the time but does not physically damage the database, such as a power failure.

- **Media failure**: A media failure is one that physically damages the database, such as a disk head crash, and affects transactions that are using the affected portion of the database.

To facilitate recovery, at various times during processing (according to some predetermined scheme) checkpoints occur. When a checkpoint is reached, any data in the system’s main memory buffers is written to the disk, and a checkpoint record is written, which comprises a list of transactions still in progress at the checkpoint.

Typically, the log will comprise two portions: an on-line portion, held on disk for immediate availability and containing relatively recent transactions, and an off-line portion, held on a backup medium and containing less recent transactions.

Concurrency

A shared database may execute transactions serially or concurrently. In serial mode, transactions are executed one at a time, and a transaction cannot begin to execute until the previous one has completed. All the resources of the database are available exclusively to each transaction, and there is no
possibility of one transaction interfering with another. This is straightforward, but is also an inefficient use of resources.

In concurrent mode, many transactions may execute concurrently, and the individual reads and writes of data which they initiate are interleaved. This uses resources more efficiently, but, in databases that use update in place (that is, those that physically change the contents of a record or a tuple) it also introduces the possibility that one transaction may interfere with another, potentially giving rise to three types of inconsistencies.

In such a database, suppose two transactions, *Give* and *Take*, both need to change your bank balance:

- **Dirty read**: *Give* reads your balance as £500, adds £200, and writes it back as £700. Before *Give* commits, *Take* reads your balance as £700. Then *Give* fails, rolls back and restores your balance to £500. The value of £700 read by *Take* is no longer true. *Take* has the wrong data.

- **Lost update**: Going on from there, having read your balance as £700, *Take* subtracts £100, writes it back as £600 and commits. Then *Give* fails, rolls back and restores your balance to £500, where it was before *Give* started. *Take* has been lost, and you have gained £100.

- **Unrepeatable read**: *Give* reads your balance as £500 but has not yet changed it. Meanwhile *Take* updates the balance to £400. *Give* now needs to re-read your balance: it finds £400 instead of £500, which is inconsistent.

The associative model’s “add only” update philosophy does not use update in place. Its commit mechanism and visibility layer ensure that one transaction may never see any part of another that has not been committed.
Security

A database needs to know who is authorised to control access to it, and who in turn has been granted authority to access to it. This is achieved by security mechanisms that allow a designated administrator to enrol users by name, and to grant authority to them to create, delete, change, use and view database objects. Authorities may be granted to users individually, or users may inherit authorities granted to groups into which they are enrolled. Broadly, users are characterised as:

- Developers, who may change meta-data, and

- Application users, who may not change meta-data but may change data. (From this standpoint, implementers may wish to treat irregular association types as data, not meta-data.)

At a more granular level, authorities may be granted over chapters, entity types, association types and queries.

Schema Modifications

One significant aspect of the associative model is that metadata describing schema and rules is stored in the database alongside data, and is maintained by the same mechanisms, so metadata is subject to the same concurrency principles as data. At a practical level this means that associative database schemas may safely be modified whilst the database is processing transactions. This may seem radically dangerous today, but, as the web continues to break down barriers of time zones and geography, the expectation of both businesses and consumers alike is that web services will be available 24 hours and 365 days, whilst still continuing to evolve and improve at the same time. At some point, databases that can evolve on-line will be needed.
Chapter 8. Logical Layer / 133

The Time Dimension

If a company whose details are stored in a relational database changes its address, when their record is updated in the database, the new address replaces the old one, and no record that the address has been changed is visible to the user. In other words, the new data overwrites and destroys the old data.

Before:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ Ltd</td>
<td>125 Canary Wharf, London E2 7YY</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ Ltd</td>
<td>Montague Court, London EC3 4RR</td>
</tr>
</tbody>
</table>

When the same operation is performed in an associative database, the association to the old address is flagged as no longer current, and a new association to the new address is created. The old association and the old address both remain in the database, and are accessible using appropriate software capabilities. No data is overwritten or destroyed.

Before:

    XYZ Ltd address 125 Canary Wharf, London E6 5TT

After:

    XYZ Ltd address 125 Canary Wharf, London E6 5TT
    ... deleted by Transaction 97756392
    XYZ Ltd address Montague Court, London EC3 4RR
    ... created by Transaction 97756392

It is also possible to provide for transactions to become effective (ie. visible) at a time later than that at which they were accepted by the database:

    XYZ Ltd address 125 Canary Wharf, London E6 5TT
    ... deleted by Transaction 97756392
    XYZ Ltd address Montague Court, London EC3 4RR
    ... created by Transaction 97756392
Unlike the relational model, the associative model does not require a separate current snapshot and historical journal (although, in the absence of full mirroring, some such capability is desirable to ensure that the database may be recovered following a physical media failure).

Because data is not changed or physically deleted, it is possible to view the state of the database, or anything in it, as it was, is or will be at a specified moment in time, past, present or future. When storage capacity and performance constraints take precedence over this capability, a standard utility may be employed to consolidate a view of the database as at a specified point in time, and to archive transaction detail prior to that point.
9. METACODE

As I said in Chapter 1, programming languages have evolved through abstraction: first, sequences of machine code instructions were abstracted into symbols to create symbolic languages, and then sequences of symbols were abstracted into third generation high level language instructions. Application development tool vendors have sought to introduce more abstraction through fourth generation programming languages (4GLs) but these have not found general favour in the marketplace, and modern programming languages such as Java and C# embody no more abstraction than their non-object-oriented predecessors such as COBOL and C. The next stage in introducing more abstraction into programming languages demands a corresponding level of abstraction in persistent data structures.

The associative model achieves this new level of abstraction. It frees programmers from the need to understand either the physical or the logical structure of data, and allows them to focus solely on the logic and processes by which users interact with databases, and databases interact with each other.

To reiterate: 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. This fact of software life is universally accepted and rarely questioned, but why is it so, and does it always have to be so?

In a relational database, every relation is structured differently. Each one has a different number of columns, and each column has a different column heading and a different domain. Because of this, programs have to be designed around the relations. Using mainstream programming languages, it is
impossible to write an efficient program that is capable of accessing a relation whose structure was not known when the program was 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 relations that it will use, and a program that uses one set of relations cannot be used with a different set.

In a typical commercial data processing application, each business entity is represented by at least one relation, and most applications involve between 50 and 500 business entities, so each new application needs somewhere between 500 and 5,000 new programs to be written from scratch. Even using advanced 4GLs and application development tools, one non-trivial program can still take days or weeks to write and test.

**Software Re-use**

The systematic reuse of existing, tested program code is the Holy Grail of software development – promising great things, always sought, but never found. Large-scale re-use was one of the key goals of object-oriented development: however, some twenty years after the first object-oriented languages were developed, almost no systematic re-use has been achieved.

(Let me be clear about what I am calling systematic reuse. Since programming began, experienced programmers have reused their own code by taking an existing program as the starting point for a new one, and by cutting and pasting existing code fragments into their new programs. I do not regard this as true systematic reuse, which demands the reuse of other programmers’ code.)

Two of the most visible attempts to make reuse a commercial reality have been sponsored by IBM. The first was Taligent, a high-profile software venture funded by IBM, Apple and HP. Taligent’s objective was to develop a definitive set of
reusable components, but the result, CommonPoint, failed to find a market and the company was absorbed into IBM. The second is IBM’s San Francisco project, a Java-based collection of components and frameworks that includes hundreds of common business objects plus application-specific components for core business processes. Commercially, the jury is still out on San Francisco, but it has been in development for some years and changed its underlying technology at least once, so the signs are not encouraging.

Some development tools automate the process of writing programs by re-using not programs but program designs. This was the route that my colleagues and I took with Synon’s Synon/2 and Obsydian¹, and some of our customers achieved productivity levels that were extraordinary by accepted measures: in excess of 1,000 lines of fully tested code per project member (developers plus testers) per day was not uncommon.

However, the initial learning curve of such tools is steeper than “traditional” third or fourth generation programming languages, because the programmer has to understand and get to know well the structure and behaviour of each of the large and functionally complex building blocks that are at their disposal. Consequently

¹ Now marketed by Computer Associates as Advantage 2E and Advantage Plex respectively.
by comparison to traditional programming techniques such tools demand a higher up-front investment in training, and slightly postpone the visible pay-back, so despite their high productivity levels, their use is not widespread, nor is it likely to become so.

The most notable success for systematic reuse has been in the field of class libraries. A class library is a set of classes, each one performing some common programming function, written in an object-oriented language and typically used within the framework of an application development tool. In the Windows environment, the Microsoft Foundation Classes (MFC) class library has become ubiquitous. This concept has also been central to Java from its inception, and much of Java’s basic functionality is delivered via the Java Foundation Classes (JFC).

Class libraries are not such a clear win for reuse as might be supposed, however. In the Windows environment, the bare Windows API (application program interface – the set of tools and procedures that allow application programs to manipulate the Windows user interface) is so complicated that C++ programming without MFC would be virtually impossible, so the MFC technology has been developed as far as it had to be to make Windows programming feasible, and very little further. Most of MFC deals with constructing the graphical user interface, and most of the rest deals with low-level programming facilities such as communications. JFC covers a broader base of functionality than MFC, but in neither case does the level of abstraction rise far above basic plumbing.

From reuse, the industry’s focus shifted in 1998 to component software. This involved re-labelling older ideas and techniques, such as DCOM and CORBA, that were in danger of exceeding their shelf-life. Also, rather like object orientation, component software strategies and tools were developed without any underlying conceptual foundation, with every vendor free to interpret the meaning of “components” to suit their own products. The true benefits of component technology will be realised only when it rises above the level of basic plumbing.
Using current database technology, components can only work together when they are designed to do so, which renders the current initiatives futile.

The bottom line is that today reuse succeeds only at the extremes of component granularity: at the bottom end of the scale through the reuse of individual classes in object-oriented languages, and at the top end of the scale through the reuse of entire application packages. In between, it is not clear that a programmer can find, purchase, learn about and tailor a component more cost-effectively than building the same component from scratch. Despite all the research funding and industry attention focussed on reuse and component technologies, most programs developed today are still hand-coded, and, even when the starting point is an existing program or a set of classes, the process is still labour-intensive.

Reuse has failed not because our programming languages and tools are deficient, or our programmers are not clever enough, but simply because we do not store our data in a way that permits it. In order to achieve the reuse of code at a level above basic plumbing, we must store data in a consistent way that allows it to be manipulated by generic, abstract programs that can be written by someone without any knowledge of how individual entity types will be structured. This “metacode” is then able to operate on any and every entity type. This approach to programming avoids the need inherent in traditional programming to specify every procedural step explicitly. One of the main objectives of the associative model is to provide a robust metacoding environment. The associative model stores metadata and data side-by-side as links: one simple, consistent format for all entities and associations. It does not need different relations for different entities, or, to look at it another way, all the entities live in the single relation with four columns that holds the links. Having removed the need for the programmer to understand the structure of every entity type, we can now write
programs that can operate on any and every 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 needed decreases still further. Most programmers today continually re-invent the wheel by rewriting familiar programs to work with new relations. Breaking this cycle will significantly reduce the cost of computing.

There is no inherent reason why metacode could not be made to work in the context of the relational model, but it would be much trickier for programmers to understand and use, because it is more difficult for a program to read heterogeneous items from many relations, each with a different number of columns, than to read homogeneous items from one relation of fixed degree. Codd specified that the relational schema – the metadata for a relational database – should be stored in a special set of tables, now generally called the catalog or data dictionary, and this is a feature of most relational databases today. However, tools that use the catalog to allow immediate, interpretive access to databases are not in general use. Metacode is not part of the relational landscape today, nor is it likely now to become so.

**Omnicompetent Programming**

As we saw 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 is stored in a database is called “metadata”. Metadata describes 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 inter-relate. Permitted 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 describes a single database is 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 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 application program that accesses a database uses a schema to tell it how the data in the database is 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 schema is not known before the program is written and 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.

Relational databases comprise varying numbers of dissimilar tables with varying numbers of dissimilar columns, and moreover their schemas are stored separately using two different languages. This makes it very difficult to write omnicompetent programs, and there is no mainstream programming environment for relational database that supports 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 types of data, and metadata as well, side-by-side in the same simple, consistent
form of items and links. 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 us to write programs 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 needed decreases still further.

Most programmers today continually 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 omnicompetent programs, many end-users will be able to develop and deploy simple applications without any specialist help.

**Separating Rules and Procedures**

If we look inside a typical unicompetent application program written in COBOL or Visual basic or Java, we will find a mixture of code that runs procedural logic, which controls how and when the program interacts with presentation devices and data storage devices, and code that enforces business rules. The goal of metacode is to separate these two entirely, leaving the procedural logic behind in the program and moving business rules into the database, where they can be expressed in codified form and made available to omnicompetent programs.
The associative model provides the capability to write omnicompetent programs that are capable of operating on any database schema without modification. What one makes of this capability is a matter of imagination and ingenuity. I present here an overview of the basic user interface that we have developed for Sentences, our own implementation of the associative model, as an introduction to what is possible.

The user interface metaphor that we have chosen is the ubiquitous two-pane explorer. We show the schema in the left-hand pane, and instance data in the right-hand pane. Figure 1 below shows a simple schema designed to record our team members, their various skills and their involvement in our current projects. The left-hand pane lists the six entity types that comprise the schema. The Person entity type is selected, and this causes instances of Person to be listed in the right-hand pane. Selection and filtering capabilities are provided on the right-hand pane to restrict the number of instances presented.
Figure 1: Explorer showing entity types and instances of Person
We can open new explorer tabs on both an individual type in the left-hand pane and an individual instance in the right-hand pane, in this way:

![Diagram showing the schema for "Person" and the Simon Williams entity.](image)

Figure 2: Schema for “Person” schema and the Simon Williams entity.

In the left-hand pane, we see the schema for the **Person** entity type. In our written notation, this is as follows:

```
Person skilled in Language
    ... expertise Level of expertise
Person involved in Project
    ... role Role
    ... hours per week Integer
```

The cardinality of both **Person skilled in Language** and **Person involved in Project** is multiple. In the right-hand pane, we see the instance data for the entity **Simon Williams**. Again, in our written notation, this is as follows:

```
Simon Williams
    skilled in, Language
    skilled in, COBOL
    skilled in, Java
    skilled in, RPG
    involved in, Sentences beta
    involved in, The Associative Model of Data
    role, Author
    hours per week, 20
    The Associative Model of Data
    involved in, Web site development
```
Simon Williams skilled in COBOL
... expertise Expert
Simon Williams skilled in Java
Simon Williams skilled in RPG
Simon Williams involved in Sentences Beta
Simon Williams involved in The Associative Model of Data
... role Author
... hours per week 20
Simon Williams involved in Web site development

Figure 3 shows the edit panel that is invoked when we select the Simon Williams entity. This allows us to amend the entity. It enforces cardinality rules, and provides selection capabilities for target entities via the selection buttons shown against each entry field (Figure 4). Selecting a type rather than an instance and invoking the same function creates a similar panel ready to capture a new instance.

From the edit panel, the shortcut menu against any of the associations allows us to invoke the edit panel for the association’s target, or for the association itself. Figures 5 and 6 show this capability in action for the Simon Williams skilled in COBOL association.

Finally, Figure 7 shows the associative model’s ability to show inverse associations. Selecting the Project entity type in the left-hand pane lists the Project instances in the right. Clicking open each in turn immediately shows us the people involved in each project, each being the inverse of a Person involved in Project association.
Figure 3: Edit panel for the “Simon Williams” entity.
Figure 4: Selection list invoked by the “…” button for “Skilled in”
Figure 5: Shortcut menu for “Skilled in COBOL”.
Figure 6: Edit panel for “Simon Williams, involved in, The Associative Model of Data”
The essential point about this series of illustrations, which is not evident from the printed page, is that all this functionality is available instantly for any and every entity type and association type that we might care to specify. The edit and search panels are assembled “on the fly” by the omnicompetent Java code that we have written to create the meta-application. The tree metaphor is particularly appropriate to the associative model. Visually, the tree is the natural representation of the associative model just as the table is the natural representation of the relational model.
10. LANGUAGE FOR THE ASSOCIATIVE MODEL

I shall discuss three distinct approaches to language for the associative model:

- a tree-based approach to querying that derives directly from the associative model;

- a set-based approach to querying offering similar capabilities to SQL that is derived from the relational model’s relational algebra;

- a document-based approach utilising XML.

Finally, I shall offer some thoughts on graphical representations of schemas.

A Tree-Based Approach

The relational model’s set-based approach to querying derives from the relational algebra and is implemented in SQL. It leads to the requirement that every element of the result set should be of the same shape. This presents a problem in manipulating data where some values are missing. The relational model addresses this by introducing the concept of null values: however, judging by the amount of debate that this feature of the relational model has provoked, it might be considered to have introduced more problems than it solved. Null values play no part in the associative model: if a particular instances has no value for a particular attribute, the association that would have linked it to an appropriate target simply does not exist.

The tree-based approach that we are about to discuss allows the shape of instances of the result set to vary as required, in the same way that the associate database allows the information held
about different entities to vary. A relational database is all about tables, and uses a query system based on tables and operations on tables. The associative database is about trees, and needs a query system based on trees and operations on trees.

Request trees

A query takes the form of a request tree, made up of request nodes. One way of looking at this is as a path through the schema. The path starts at a suitable point, and by taking one step at a time, each time following some association type, in either direction, arrives at the data needed for a particular query. An entity type or association type is initially selected to form the root of the request tree, like this:

```
Example

  Employee

Parameters
```

The root of the request tree is known as the data request node.

The next step is to decide which association types are required, and add the appropriate request nodes. Any association type whose source is Employee, or a subset or supertype of Employee, can be added as a forward request, which returns association instances in the forward direction. Our example query could be extended like this:

```
Example

  Employee

      date of birth, Date

      home telephone, Telecos number

Parameters
```

Similarly, an association type whose target is Employee or a related type can be added as an inverse request, which returns association instances in the reverse direction, that is traversed from target to source. For example:
To extend the request tree further, any request node can have forward or inverse request nodes added to it, provided the corresponding association type matches the node it is being attached to. For example, the **Employee** event **Date** association type is the source of other association types, allowing us to build a two-level query like this:

A forward request node always has a child node, known as a *target request*, representing the target of the corresponding association type. In the example above, there is a node marked “Date” which is the target request belonging to the forward request immediately above it. Similarly, an inverse request has a *source request* as a child node. The purpose of these request nodes is to provide somewhere to attach further request nodes. Using this, the path through the schema can be as long as necessary, like this:
Result Trees

To execute a request tree, instances of the entity type or association type corresponding to the data request are retrieved and processed. For each instance, the immediate child nodes of the data request are retrieved, and used to further retrieve a set of associations which are added to the growing result tree. This recursive process is carried out to any depth required. The result is a tree of data whose shape is derived from, but not identical to, the request tree. This diagram shows an example of a request tree and a corresponding result tree:
Note that the root node of the result tree is not shown. The diagram shows a single main branch of the tree, corresponding to a single instance of the Person entity type. Each request node generates one or more sets of result nodes; each result node is derived from one and only one request node.

**Binding Nodes**

The default action when executing a request tree is to return all instances of the type that the data request node refers to, and to build a branch of the result tree from each one. One way to limit the amount of data returned is to tell the data request node that you are only interested in a specified instance of this type. This is known as binding the node. Binding the data request node builds a result tree with only a single main branch, based on the instance specified. Executing this query:
returns a list of all employees, with their dates of birth and telephone numbers. Binding the data request node turns the query into this:

```
? Example
  - ⚫ Employee [Harold Ferry]
    - 🔴 date of birth, Date
    - 🔴 home telephone, Telecoms number
  - Parameters
```

which generates this result tree, with only a single branch:

```
- ⚫ Harold Ferry
  - date of birth, 15-Apr-1977
  - home telephone, (210) 182 0932
```

Binding is not limited to the data request node. Any node anywhere in the request tree may be bound, which will cause that node only return data that matches the instance it is bound to. For example, let’s take a query on Employee, which returns data about employment events. To restrict the results to events which took place on a particular date, we can bind the target node “Date”, like this:

```
? Example
  - ⚫ Employee
    - 🔴 event, Date
      - Date [01-Jan-1998]
      - event, Event
      - job title, Job title
  - Parameters
```

The result tree looks in part like this:
There is a branch for Barney Norris, although he has no events on the specified date. Since there are no suitable events, none are shown. Bill Jupiter’s branch shows an event matching the date specified. Other events for other dates are simply not shown.

To suppress employees who have no events on the date in question, the request tree may start from the association type `Employee` event `Date`, like this:

```
Example
```

which produces results like these:

```
(Bill Jupiter, event, 01-Jan-1998)
  event, Joined
  job title, Systems Architect

(Freda Quentin, event, 01-Jan-1998)
  event, Joined
  job title, Programmer
```

There are three ways of binding a node:
• It can be bound to a fixed instance, as shown in the examples above.

• It can be bound by a query parameter, as described below.

• It can be bound to another node, to perform something like a relational join operation.

As an example of the third case, consider this query:

Here, a person has visited a store on a given date, and bought a number of products. We want to know the prices of those products. The query execution proceeds normally down the request tree as far as the Product node, and then follows an inverse link to Store to retrieve the price. As it stands, this query will return all prices charged for each product in any store, not just the store where it was bought that day. To restrict the results to only the prices charged at the particular store where this person went shopping, we need to bind the second Store node. However, this binding is not fixed. For each visit, we simply bind the second Store node to the corresponding store for that visit, which is the current result from the first Store node.
Query Parameters

Permanently binding a node to a fixed instance is of limited use. For example, with the query for employment events on a given date, you probably want to re-run the query for a number of different dates. This could be done by editing the query each time, but it is much easier to define a query parameter. A query parameter has a value which is different each time the query is executed. When a query with parameters is exercised, the user is prompted to enter a value for each parameter. Here is an example of a query with a parameter:

The parameter is called “Date”. When the query is executed, request nodes called “Date” will be bound to the instance whose value is supplied for the parameter.

A query parameter has a name and a type. The type can be an entity type or association type, and must correspond to the request node that is to be bound. Therefore, to bind more than one node with a single parameter, each must refer to the same entity type or association type. The only nodes that are bound by a parameter are nodes with the same name as the parameter. The names of request nodes and parameters may be changed at will to achieve a desired result.
For example, if the employment events example shown above formed part of a larger query where other dates appeared, you could ensure that the parameter binds only one node by changing the example like this:

![Diagram]

Each parameter has the following attributes:

- **Name**: the name used to refer to this parameter value
- **Type**: the entity type or association type whose instances this parameter represents
- **Default Value**: the value that is taken when no other value is specified. The parameter in the example above has no default value, so that if the user does not supply one, the parameter has no value, and its matching node remains unbound. If a default value is supplied, then the matching node is always bound.
- **Mandatory**: a mandatory parameter must have a value. If it does not have a default value, then the user must supply one, or the query cannot be executed.

**Operator Nodes**

The results returned by a query may be further modified by adding *operator requests* to the request tree. An operator request can:

- specify calculations to be carried out using the query results (a *derived type* request);
• restrict the results returned by another request node, by applying a filter condition (*selection* request);

• cause the results returned by another request node to be sorted (*sort* request).

Every operator request has an expression associated with it, which defines the computation or filtering to be carried out. In addition, the effect of an operator request depends on the point where it is attached to the tree. An operator request affects the results generated by its immediate parent, as shown by these examples.

### Derived Types

A derived type request causes new values to be generated. One value is generated for each result instance produced by the derived type’s parent. This example counts the number of people assigned to each project:

```
? Example
  ⬤ Project
  └── resources, Company resource
        └── Company resource
            └── Number of people: Count(Company resource)
                └── f(x) Count(Company resource)

Parameters
```

The node *Number of people* is a derived type request. Its parent is the *Project* node, so that it will generate one new value for each result from the *Project* node, that is, for each project. The value it generates is a count of the number of *Company resource* results which appear as children of that *Project* result, like this:
To get an overall total of the number of people assigned to projects, we could either move the **Number of people** node to the query node, or create a new derived type node, in order to see both levels of totals at once, like this:

```
? Example
- Project
  - resources, Company resource
    - Company resource
  - People assigned to this project: Count(Company resource)
    - \( f(x) \) Count(Company resource)
  - People assigned to projects: Count(Company resource)
    - \( f(x) \) Count(Company resource)
```

The partly-expanded result tree looks like this:

```
- CRM application development
  - resources, Company resource
    - People assigned to this project, 3
  - ERP prototype
    - resources, Company resource
      - People assigned to this project, 6
```

**Selection**

Results returned by a request node can be filtered by the addition of a selection request. The expression associated with a selection request represents a condition that must be true for each result instance to be returned. For example, this query selects all employment events that occurred in 1998:
Part of its result tree looks like this:

There is a result node for Barney Norris, even though none of his employment events took place in 1998. The selection request only affects the results of its immediate parent. We can change the query not to return employees who do not have recorded events in 1998 by adding a second selection request, like this:

This uses the first selection request to filter Employee event Date associations, and then uses the results as data for the second
selection request. Its result tree looks exactly like the previous example, except that the Barney Norris result is not shown, along with others that do not have employment events in 1998.

Sorting

By default, a request node returns results in the order that they are returned by the database. Entity instances are sorted; association instances are either sorted or sequenced, depending on the properties of the association type. You can change this behaviour by adding a sort request to the request tree. Like a selection request, it operates on the results returned by its immediate parent. If the immediate parent of a sort request is the data request node, the main branches of the result tree are re-ordered, as in this example, which lists employees by age, youngest first:

![Diagram of a sort request tree]

Part of the result tree looks like this:

![Employee tree with sorted dates]

A sort request further down the tree re-orders the result nodes within each set, rather than imposing an overall order.
Hiding Branches

The expression used by a derived type or a selection request can only use data that is retrieved as part of the query. This leads to including certain data in the query just so that an expression can use it; you do not necessarily want to display it. A good example is counting the number of people assigned to each project. The example query shown above includes the list of people assigned, as well as the number of people. To suppress display of the individual names, the Project resources Company resource node may be marked as hidden, like this:

![Example diagram]

If you hide a node, all its children are hidden as well. The result tree from this query looks like this:

![Recursive Closure example diagram]

Recursive Closure

Some relationships between real-world entities are transitive, which means that they carry over from one entity to another. An example is “ancestor of”: if Henry is an ancestor of Clare, and Clare is an ancestor of Edward, then Henry is also an ancestor of Edward. If we have a database containing associations Person parent of Person, then we can use recursive closure to generate a
result tree containing all the known ancestors of a given individual. The query looks like this:

```
? Example
  Person
  has parent, Person
  Parent Closure to Person ...
Parameters
```

It generates a result tree that looks in part like this:

```
  BS
  has parent, Person
  has parent, FS
  has parent, Person
  has parent, TS
  FS
  TS
  has parent, Person
  has parent, AS
  TS
  AS
  FS
  has parent, Person
  has parent, TS
  TS
  AS
```

The recursive closure operation creates a loop in the request tree. This loop acts something like a wheel, rolling through the data, generating a new level of the result tree with each turn. This is what the term “recursive” means. However, if there is a loop in
the data, which is unlikely with the “parent of” example, but can happen in other cases, then the wheel could roll forever: to avoid this, each instance is checked to make sure that it has not already been processed. If it has, then it is not processed again. This is the “closure” part of the term “recursive closure”.

A Set-Based Approach

At its most fundamental level, the relational model deals with sets: a relation is a combination of sets and the relational algebra, as we saw in Chapter 3, defines a number of set-based operations that may be performed on relations. SQL is the standard language for specifying a group of operations to be performed on a group of relations to produce a desired result. In the relational model, SQL functions as both a data definition language for schema definition, and a data manipulation language for update and retrieval.

Here, I discuss an associative algebra, derived directly from the relational algebra, and its application to querying associative databases.

Associative Algebra

The associative algebra is derived directly from the relational algebra. The operators operate on types, which may be entity types or association types.

- The **union** of two types forms a third type containing all the instances that appear in either or both of the two types.

- The **intersection** of two types forms a third type containing only the associations that instantiate both the two types.
• The **difference** of two types forms a third type containing only the instances that instantiate the first and not the second type.

• The **product** of two types forms a third type which is instantiated by all possible associations of instances of the first type as source together with instances of the second type as target.

• **Select** defines a subset of the instances of the original type.

• **Project** forms a type containing a sub-tree of the original type.

• The **join** of two types forms a third type having the first and second types as source and target respectively, where source and target instances share a common instance of a sub-tree.

• **Divide** operates on two types and forms a third, whose type is the source of the first type and whose instances are the sources of instances of the first type that are associated with all the instances in the second type as targets.

• **Extend** forms an association type that has the original type as source and a new type, instances of which are derived from the source, as target.

• **Summarise** forms a type whose instances are formed by grouping together instances of the original type that have the same sub-tree as source, and creating one instance of the new type for each such group, with the sub-tree as source and an instance aggregating corresponding sub-trees of the group as target.

• **Rename** forms a type by renaming one or more of its sub-trees.
• **Recursive closure** forms a relation by joining a self-referencing type with itself, taking the result and joining it again with the original type, as many times as necessary.

**Closure**

The associative algebra treats entity types and association types as the same thing: in effect, they are both subclasses of type. This does not compromise the closure property in the associative algebra, because every operation on a type produces another type. In this context, an entity type may be regarded as analogous to a relation of degree 1, and an association type analogous to a relation of degree > 1. Two of the relational operators are sensitive to degree: divide operates only on one unary and one binary relation, whilst project is meaningless (or at least is capable only of producing an identical relation or an empty relation) on relations of degree 1. This is not held to compromise the closure property of the relational algebra.

**Implementation**

According to Frost in [20], the mechanism for querying a triple store was first described by Feldman in a technical note called “Aspects of Associative Processing” [16] written at the MIT Lincoln laboratory, Lexington in 1965. He used the term ‘simple associative forms’ to describe the seven ways in which triples could be retrieved:

\[
\begin{align*}
(a, b, ?) & \quad (a, ?, c) & \quad (?, b, c) \\
(a, ?, ?) & \quad (?, b, ?) & \quad (?, ?, c) \\
(a, b, c) & \\
\end{align*}
\]

Each returns a set of triples whose source, verb and target match the values supplied as ‘a’, ‘b’ and ‘c’. Other authors have discussed the concept, including Levien and Maron in “A Computer System for Inference and Retrieval” [17]; Feldman and Rovner in “An ALGOL-based Associative Language” [18];
Sharman and Winterbottom in “The Universal Triple Machine; a Reduced Instruction Set Repository Manager” [19]; and Frost in “Binary Relational Storage Structures” [20]. Sharman and Winterbottom also point out similarities with Prolog.

The simple associative forms remain the foundation of querying under the associative model, but they are leveraged and extended with a number of additional mechanisms. In particular, the fundamental capability under the associative model for the source or target of an association (ie. triple) to be another association significantly increases the expressive power of the associative model, and hence the scope and sophistication of queries that may be performed over an associative database. Our own implementation of an associative database uses a grid file to store items and links, accessed by an R-tree spatial index. See Nievergelt, Hinterberger and Sevcik, “The Grid File: An Adaptable, Symmetric Multikey File Structure” [21], and Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching” [22].

**Querying**

The associative model has set-based query capabilities equivalent to those of the relational model. We will consider some examples using the associative algebra. For practical purposes, a language is required: SQL can be readily adapted to the associative model, and has the advantage of familiarity, so we will use it.
This schema reads as follows:

- **Person** customer of **Store**
  - ... visited on **Date**
  - ... bought **Product**
    - ... times **Quantity**
- **Store** sells **Product**
  - ... at **Price**
- **Product** belongs to **Category**

Here is some data:

- **Mary** customer of **Safeway**
  - ... visited on **25-Sep-99**
  - ... bought **Cornflakes**
    - ... times 2
  - ... bought **Coffee**
    - ... times 1
  - ... bought **Yoghurt**
    - ... times 6
  - ... visited on **1-Oct-99**
  - ... bought **Milk**
    - ... times 3
- **Bill** customer of **Tesco**
  - ... visited on **3-Oct-99**
  - ... bought **Yoghurt**
Safeway sells Cornflakes... price £1
Safeway sells Coffee... price £2
Safeway sells Milk... price £0.60
Safeway sells Yoghurt... price £0.30
Tesco sells Yoghurt... price £0.32
Cornflakes category Cereals
Coffee category Beverages
Milk category Dairy
Yoghurt category Dairy

Now, some queries:

An easy one first. Who shops at Safeway?

Q: Select (Person customer of “Safeway”)
A: Mary customer of Safeway

Now a little more complex. What dairy products has Mary bought?

Q: Select (((“Mary” customer of Store) visited on Date) bought Product) join (Product category “Dairy”))
A: (((Mary customer of Safeway) visited on 25-Sep-99) bought Yoghurt) join (Yogurt category Dairy)
   (((Mary customer of Safeway) visited on 1-Oct-99) bought Milk) join (Milk category Dairy)

This example uses the join query operator. When two types are joined, a new association type is formed:

<Type 1> join <Type 2>

When the query is executed, instances of this association type are created. These instances do not persist in the database, but are created afresh each time the query is executed. They are called virtual associations. We shall discuss the precise implementation of join later, but broadly the creation of a virtual
association is prompted by the occurrence of the same instance on both sides of the join prescription. In the example above, the virtual associations are created on **Yoghurt** and **Milk** respectively.

Now some calculations. How much did Mary spend (in any store) on 25-Sep-99, and how many products did she buy? This is quite a complex query, so we will split it into its various stages. The first step is to determine how much of what Mary bought on 25-Sep-99, which is the query:

Q: \[
\text{Select} \left( \left( \text{"Mary" customer of } \text{Store} \right) \text{ visited on } \text{"25-Sep-99"} \right) \text{ bought } \text{Product} \text{ times } \text{Quantity} \]

A: \[
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought Cornflakes times 2} \\
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought Coffee times 1} \\
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought Yogurt times 6} \\
\]

Next we need to know prices, so we join the result of this query with another:

Q: \[
\text{Select} \left( \left( \text{"Mary" customer of } \text{Store} \right) \text{ visited on } \text{"25-Sep-99"} \right) \text{ bought } \text{Product} \text{ times } \text{Quantity} \text{ join } \left( \left( \text{Store sells Product at Price} \right) \right) \\
\]

A: \[
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought Cornflakes times 2} \text{ join } \left( \left( \text{Safeway sells Cornflakes} \right) \text{ price £1.00} \right) \\
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought Coffee times 1} \text{ join } \left( \left( \text{Safeway sells Coffee} \right) \text{ price £2.00} \right) \\
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought Yogurt times 6} \text{ join } \left( \left( \text{Safeway sells Yogurt} \right) \text{ price £0.60} \right) \\
\]

Now we are ready to do some calculations, using the **Group** operator, and defining some expressions:

Q: \[
\text{Summarise } \left[ \text{"spent"=Sum(quantity*price);"bought products"=Count{}}, \text{ Select } \left( \left( \text{"Mary" customer of } \text{Store} \right) \text{ visited on } \text{"25-Sep-99"} \right) \text{ bought } \text{Product} \text{ times } \text{Quantity} \text{ join } \left( \left( \text{Store sells Product at Price} \right) \right) \right] \\
\]

A: \[
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ spent £7.60} \\
\left( \left( \text{Mary customer of Safeway} \right) \text{ visited on 25-Sep-99} \right) \text{ bought products 3} \\
\]

In each of our examples, the result set comprises associations. What if we want a set of entities instead? Going back to our first
example, “Who shops at Safeway?”, we might have preferred to write:

Q:  *Project Person from Select (Person customer of “Safeway”)*
A:  Mary

We can then go on to operate on sets of entities with set operators.

Who shops at either Safeway or Tesco?

Q:  *Union (Extract Person from Select (Person customer of “Safeway”)) and (Project Person from Select (Person customer of “Tesco”))*
A:  Mary
    Bill

Before we leave our examples, let’s take a closer look at the virtual associations created by join and certain other query operators. It is useful to envisage these as association types in the diagram. Here is the join that we created to discover what dairy products Mary had bought:
And here is the join that we created to work out what Mary had spent:

![Diagram showing the join between Person, Store, Date, Product, Price, Category, with arrows indicating relationships such as customer of, sold at, bought on, etc.]

**Variables**

In the examples above, the names that appear are those of entity types: in the query `Person lives in "America"`, `Person` is the name of the entity type that is the source of the `Person lives in Country` association type. However, in queries, such names are not entity types, but variables. When a query is executed, its variables are populated by result sets. So when we execute the query `Person lives in "America"`, the variable `person` becomes populated by the instances of the entity type `Person` that have `lives in "America"` associations. Variables are the mechanism that allow the members of a result set to be referenced at higher levels in the query.

The underlying type of a variable is always known from the association type that was used to define the query. This means that variables can be renamed without altering the definition of the query. So instead of `Person lives in "America"`, we could have written `American lives in "America"`. Entity types do not need to be
renamed in order to behave as variables: the type names themselves also act as variable names. Variable names are introduced to add meaning.

**XML: A Document-Based Approach**

As I mentioned earlier, XML is the abbreviation and common name for Extensible Markup Language, an open standard for describing data, defined by the World-Wide Web consortium (W3C). Like HTML, XML is a subset of SGML (Standard Generalized Markup Language), an ISO standard meta-language for defining the format of text documents. XML defines data elements on web pages and in documents, and provides a way (as it appears now, rapidly becoming the de facto standard way) of identifying and transmitting data via the web. The whole area of XML and web services is developing rapidly at the time of writing, so some of what I say here may be out of date.

XML provides a set of rules for laying out documents, and a means to define further rules which constrain the documents which are acceptable for a particular purpose. A document is well-formed if it meets the basic rules laid down by the XML standard; in fact, a document which is not well-formed cannot truly be said to be an XML document at all. An XML document contains one or more elements, where an element is either text delimited by a start-tag and an end-tag, or an empty-element tag by itself.

Here is an example of a start-tag:

```
<Employee>
```

It consists of a name enclosed in pointed brackets (less-than and greater-than signs). The name is the name of an element type, and must match the name in the corresponding end-tag, which looks like this:

```
</Employee>
```
Everything between the start-tag and the end-tag is the **content** of the element, which in general may consist of a mixture of text and other elements. Elements can be nested to any depth, but they cannot overlap in any other fashion. The primary rule of well-formed XML is that every start-tag must be matched by the corresponding end-tag, and that any element started between them must also finish between them.

To make a well-formed document, any element type names and attribute names may be freely chosen, subject to certain grammatical rules. However, to ensure that a document is a usable HTML page, for example, or an acceptable purchase order, further constraints may be placed on the element types, their names, attributes, and content. This is done by creating an XML schema, itself an XML document. (Since the publication of a W3C standard in May 2001, XML schemas are superseding Document Type Definitions (DTDs), an earlier mechanism for defining the same thing.

Extensible Stylesheet Language (XSL) is a language for visually rendering XML documents. It consists of two parts:

- XSL Transformations (XSLT): a language for transforming XML documents.
- An XML vocabulary for specifying formatting semantics (XSL Formatting Objects).

An associative database can readily generate XML documents based on queries defined using the tree-based query method described in the preceding section. The structure of the document reflects the result structure of the query.

Importing data from XML documents can be done in a similar way. The documents to be imported must be valid documents conforming to the XML schema or DTD generated from a tree-based query.
Associative versus Relational

The associative model is better suited than the relational model to a world in which XML is becoming the native language for data exchange, for two reasons: one structural, another practical.

Because of its heavy reliance on the primary key/foreign key mechanism, the relational model can be characterised as a “many-to-one” architecture: that is, many sales order lines point to – ie. carry the foreign key of – their parent sales order; many sales orders similarly point to their parent customer. XML, by contrast, comes from a document heritage. Both documents and human beings naturally work in a “one-to-many” way: first the customer, then the customer’s sales orders, then for each order the sales order lines. This is also the natural pattern for associations in the associative model: in common with XML and humans, it too is most comfortable in the “one-to-many” world.

The relational model also runs into trouble in ordering data. There is no provision in the relational model to apply any order to columns other than that in which they are defined in their SQL DDL (and indeed, Codd expressly proscribes imputing significance to the order in which columns occur in base tables). This means that data contained in an XML document that is added to a relational database and subsequently extracted is unlikely to emerge in the same order in which it went in. The
associative model, by contrast, includes native capabilities to
give users express control over the order of both association
types and associations.

**Graphical Design and Modelling Techniques**

Many graphical design and modelling techniques, representations and methodologies have been proposed over the years, from decision tables and flowcharts in the 1960s to UML and ORM today. There is a fundamental problem with their use, which is this. Modelling techniques that entail a complex transformation between conceptual and logical models (ie. ORM to relational tables) add little value to the overall development process, because such transformations are inherently forwards-only: once the conceptual design (an ORM) has first been transformed into a logical design (an SQL schema), all subsequent design work must be done at the logical design level, because changes to the logical design cannot be backed into the conceptual design without loss of information.

This means that such techniques are really only useful in the early conceptualisation of an application, and are of little use thereafter (except as documentation if and only if they are rigorously maintained by hand, which they seldom are). It follows that they have a role only in waterfall developments, and are counter-productive in rapid, iterative development, because the latter acknowledges that the first-cut conceptual design is very far from the final objective.

At Synon, we overcame this by providing a conceptual modelling environment from which the logical design could always be generated automatically in its totality. To achieve this, we had to provide conceptual constructs to model implementation-level features of the logical design, but this entailed a lot of complexity.
The associative model overcomes this problem by providing conceptual, logical and physical models that are unified with each other and with reality: in other words, an entity in the real world is directly represented by an entity in an associative database, which is directly represented by an item on disk; and an association in the real world is directly represented by an association in an associative database, which is directly represented by a link on disk.

This having been said, clearly those who find the use of graphical modelling techniques helpful, in whatever context, should be free to use them with the associative model. However, there is one important consideration. The associative model is almost unique amongst modelling techniques in that it allows association types to have attributes in their own right, without insisting that they should first be represented as entities, or “reified”. Reification is a technique described in W3C’s RDF Model and Syntax Specification:

“In order to make a statement about another statement, we actually have to build a model of the original statement; this model is a new resource to which we can attach additional properties.” ... “This process is formally called reification in the Knowledge Representation community. A model of a statement is called a reified statement.” ... “A statement and its corresponding reified statement exist independently in an RDF graph and either may be present without the other.”

Reification is at odds with the Associative Model. As the extract makes clear, RDF demands that a statement which is the subject of another statement must be represented twice in RDF: once in its original form and a second time in its reified state. This is a redundant and unnecessary overhead that also raises significant integrity issues. Under the associative model, a statement does not need to be reified in order to become the subject (or object) of another statement.
Reification is also a fundamental aspect of the relational model, in that any table that has a foreign key as a primary key is a reified representation of an association.

Graphical modelling tools for use with the associative model must not insist on reification. In practical terms, this means that they must allow an arc to join two nodes, or a node and an arc, or two arcs.
11. BUSINESS RULES

Shamefully late in the day, considering the amount of money that has been invested in application development over the past four decades, the software industry is beginning to think about giving managers more control over systems by automating business rules. I said in Chapter 9 that the goal of metacode is to separate procedural logic from business rules and move the assertion and enforcement of business rules into the database management system. In this chapter I shall examine some of the work done on business rules so far, and consider how rules may usefully be codified and recorded in an associative database for use by omnicompetent programs.

Terminology

According to the Concise Oxford English Dictionary, a rule is a principle to which an action conforms or is required to conform. The term “business rule” has been adopted into industry parlance and is used by industry groups such as the Business Rules Community and the Business Rules Group. However, the word ‘business’ is not very helpful or even precise. Many organisations who use business rules, such as governmental bodies, are not businesses in the accepted sense. Also, when discussing rules, it is often important to make the scope within which the rule applies clear, and the most convenient way to do this is with a prefix: school rules, club rules, house rules, state rules and so on. But given that, by its nature, the work of IT practitioners lies exclusively within “businesses”, ‘business’ as a useful scope for rules is redundant.

Moreover, as we shall see, we need our own set of scopes for rules – presentation rules, domain rules and so on – so I propose to drop the word ‘business’ and talk simply about rules. I shall also use two made-up words, ‘edictor’ and ‘edictee’, to
mean those that respectively make and obey rules. I shall refer to companies, organisations, partnerships etc as ‘enterprises’.

**Rules in an Enterprise**

The edictors who make the rules that shape an enterprise are many and varied. There are outsiders: local, national and international legislatures, trade and regulatory bodies, and markets, both local and global. There are insiders: shareholders, corporate officers, executives, managers, staff and affiliated companies. There are associates: customers, suppliers and advisers. And the people within an enterprise are themselves subject to rules set out by morality systems, religious beliefs and their own experience and instinct.

Are all the rules made by all these edictors relevant to humble database applications? The answer is unequivocally yes. Just a few examples: In the UK, law firms must observe strict rules set by their trade body, the Law Society, concerning their handling of clients’ funds, and these rules must be closely monitored by their client accounting systems. Also an application that raises invoices for non-existent deliveries would put its enterprise at risk of breaking the law. And an enterprise with Muslim employees and a sensitive human resources policy would not schedule a company picnic during Ramadan.

To repeat, the OED’s definition of a rule is a principle to which an action conforms or is required to conform. This simple and robust definition seems to accord well with our intuitive understanding. It also seems to be intuitively correct (though I am not attempting to prove it here) that the essence of an enterprise is the sum of all its actions and possible actions. So if we could express all the actions that an enterprise has taken and might take, and all the rules that governed and will govern those actions, we would have a perfect model of the enterprise. (I’m not proposing this as a practical exercise: one of the most
influential edictors in any enterprise is its chief executive, and he or she is likely to be subject to his or her own unfathomable edictors such as experience and instinct.)

**Good Rules are Declarative**

A good rule makes assertions about states, not about processes, and leaves edictees in no doubt about whether they are complying with the edictor’s intention. For example, “*Keep off the grass*” is a bad rule. It is clearly open to violation: if someone sees a child in danger on the grass, they will have to break the rule to help the child. Also, the edictor’s intention is unclear: the rule doesn’t preclude damaging the grass in other ways that don’t violate it, such as throwing strong chemicals over the grass or driving cattle onto it. The problem is that the rule is expressed in terms of a process – “*proceed in a way that does not require you to walk on the grass*” – not a state, and so it fails to explicitly state what the edictor is trying to achieve.

To express this rule properly through processes, the edictor would have to say “*Keep of the grass (unless you are a gardener, or in an emergency), do not throw things onto the grass (except grass food and water in appropriate quantities and at appropriate times of the year), do not allow large animals to run on the grass, do not...*”. A better expression of this rule would be “*Do not damage the grass unnecessarily*”. This confines itself to describing the state that the edictor wishes to prevent – having the grass unnecessarily damaged – and so avoids having to expressly prohibit all of the many possible processes that might give rise to this state.

This example illustrates that rules are better stated declaratively rather than procedurally. A declarative rule sets out a state or states to be achieved or avoided, whilst a procedural rule sets out procedures which will, when correctly followed,
result in a state or states being achieved or avoided. Why is declarative better? Here are some reasons:

- It may often be difficult to tell whether a procedural rule has achieved its desired outcome. For example, directions to a location may be given procedurally, as in “Take the M40 west to junction 3, turn south and go to the first building on the right past the second traffic lights”; or declaratively, as in “Go to 25 High Street, Burnham”, which allows you to use a map (another set of declarative rules) to find your way by the most effective route. Only the declarative rule allows you to know unequivocally that you have complied with the edictor’s intention by arriving at your destination.

- Procedural rules need to be tested and maintained on a continual basis. Plenty of things can go wrong when you try to follow the procedural directions. You may be starting from west of Junction 3, or a third set of traffic lights may have been installed, or a more direct route may have been constructed.

- Procedural rules are almost always derived from declarative rules, with a consequent loss of information. The edictees of procedural rules are less well-informed and consequently less motivated. A child who is told “Brush your teeth every night” is ignorant of the consequences of non-compliance, so is less inclined to comply than one who is told “Prevent your teeth from decaying and falling out”. A workforce governed by procedural rules will be less well-informed and have less scope for initiative than one governed by declarative rules.

- Procedural rules are less implementation-independent than declarative rules, in that they are usually framed for a particular mode of process. For example, the procedural directions to a location would have to be different depending on whether the main vehicle to be used was a car, the
railway system or a helicopter. The declarative version works for all three. Implementation independence is a vital quality in today’s fast-moving technology world.

Most significantly, procedural rules in software applications have to be implemented by procedural code, so type-specific coding must be written, tested and maintained for each rule. A declarative framework for rules, in conjunction with generic data structures such as those provided by the associative model, offers the opportunity to write abstract, type-generic code that is capable of applying business rules across all types within a database.

**Rules So Far**

To get a better sense of what rules are all about in the context of database applications, let’s consider the work that has already been done.

**Martin and Odell**

James Martin and James J. Odell in “Object Oriented Methods: A Foundation” [37] set out this taxonomy for rules:

```
Rules
  ├── Constraint Rules
  │       └── Operation Constraint Rules
  │       └── Structure Constraint Rules
  └── Derivation Rules
        └── Inference Rules
            └── Computation Rules
  └── Stimulus/Response Rules
```
Let’s get a sense of what each type of rule means. The words are theirs and are italicised; the examples are mine.

- **Stimulus/response rules** constrain behaviour by specifying *WHEN and IF conditions that must be true in order for an operation to be triggered*. Eg. If a sales order item line has not been packed and goods are on hand, produce a packing note.

- **Operation constraint rules** specify *those conditions that must hold before and after an operation to ensure that the operation performs correctly*. Eg. An unfulfilled sales order cannot be billed.

- **Structure constraint rules** specify *policies and conditions about object types and their associations that should not be violated*. Eg. A sales order must have at least one item line.

- **Inference rules** specify *that if certain facts are true, a conclusion can be inferred*. Eg. If all item lines have been shipped or cancelled, then the order is fulfilled.

- **Computation rules** derive results by processing algorithms. Eg. Sales order line value = Quantity ordered X Item price.

Let’s deconstruct Martin/Odell’s taxonomy and see where it leads us. Firstly, are operation and structure constraint rules essentially different? Only if one allows that something other than an operation may alter structure. If we prohibit this possibility by saying that operations are the only things that may alter structure, then we can unite them. So all structure rules are operation rules, because only operations may alter structure, and the only time that it is meaningful to test structure rules is at the time of an operation, because if no operation has occurred, structure cannot possibly have altered. The definition of
“operation” must of course include the creation and deletion of instances.

Secondly, how different are inference and computation rules? Martin/Odell themselves say “Computation rules can be thought of as inferences. The primary difference, however, is the manner of expressing the derivation. The inference rule is a rule conceived in an IF ... THEN manner. The computation rule is a rule conceived as an equation.” Thus, at the conceptual level, we can regard them as the same thing.

The authors also go on to say “Computation rules can appear to be just another structure constraint rule, because both have an IT MUST ALWAYS HOLD quality. ... To a user, they describe different ways of thinking about and specifying business rules.” Again, we’ll see how we fare conceptually by regarding them as the same thing.

So, conceptually and for the time being, we can simplify Martin/Odell’s taxonomy from five types of rule to two: stimulus/response rules and structure constraint rules, with the proviso that both have access to computation and inference capability.

**Ronald Ross**

In “The Business Rule Book: Classifying, Defining and Modeling Rules” [40], Ronald G. Ross takes a more analytical approach. He comments “Rule types can be viewed as the alphabet for a rule language (or more precisely, for a sub-language for integrity)”. Ross is the creator of the “Ross Method”, an approach to business rules that embodies detailed taxonomies for both atomic rule types (those which cannot be derived from combinations of other rule types) and derivative rule types (those that may be expressed by specification of other rules), together with a graphical syntax for expressing rules. Ross identifies seven families of atomic rule types, comprising 33 rules types in all. The seven families are:
• **Instance verifiers**: deal with the number of instances of B associated with each instance of A.

• **Type verifiers**: deal with whether A and B are mutually exclusive, inclusive or prohibited.

• **Position verifiers**: deal with innate sequence amongst instances of A.

• **Functional verifiers**: deal with inter-relationships amongst instances of A (eg. always ascending, never repeating).

• **Comparative evaluators**: deal with equality, inequality, less than and greater than between A and B.

• **Mathematical evaluators**: deal with arithmetical operations involving A and B.

• **Projection controllers**: deal with projecting values from A to B.

Ross also identifies twelve families of derivative rules, comprising 58 types of derivative rule in all. The families are:

Testing instances
Verifying position
Modifying attributes
Controlling sequence
Specifying successions
Testing composition structures
Evaluating relative time
Evaluating updates
Co-ordinating activity
Enabling instances
Copying instances
Invoking instances

Ross’s work is very fine-grained: he defines and discusses 19 families and 91 types of rules, together with many real-world
examples using his graphical notation. I make no judgement on the accuracy of Ross’s taxonomy; however, I fear that it is too arcane for the mainstream. Also it remains to be proven to what extent tools based on Ross’s work might help developers to improve their efficiency by making the application development process more amenable to automation.

Finally, for the reasons discussed in Chapter 10, graphical techniques are really only useful in the early conceptualisation of an application.

**Chris Date**

In “What Not How: The Business Rules Approach to Application Development” [39], Chris Date classifies rules into Presentation rules, Database rules and Application rules. He concedes that it is hard to draw a crisp distinction between database and application rules, and treats them together. Date discusses Martin/Odell’s taxonomy, modifying it slightly:

```
  Rule
    Constraint
      State Constraint
      Transition Constraint
    Stimulus/Response
    Computation
    Inference
```

Date also defines an alternative taxonomy for constraints (which, as he points out, are known more specifically as integrity constraints) that is better aligned with the relational model of data. Date classifies constraints as follows, using his words (italicised) and my examples:
• **Domain constraints** specify the legal values for a given domain. Eg: The domain **Quantity ordered** contains all integer values between 1 and 99.

• **Column constraints** specify the legal values for a given column. A column constraint for some given column C is a statement to the effect that values of that column are drawn from some given domain or type D. Eg: Values in the **Quantity ordered** column of sales order item lines must be drawn from the **Quantity ordered** domain.

• **Table (or relvar) constraints** specify the legal values for a given table. A table constraint can be as complicated as you like, provided only that it refers to the individual table in question and no others. Eg: if a customer’s status is Good, its credit limit should be at least £1,000.

• **Database constraints** specify the legal values for a given database. A database constraint can be as complicated as you like provided only that it refers to at least two distinct tables. Eg: if a customer’s status is Poor, the value of its orders may not exceed £500.

Date says that stimulus/response constraints are “really just triggers which open a door to ‘creeping procedurality’”. Whilst not proscribing them outright, he counsels that they should be used sparingly. In my view, Date goes too far. Whilst he is right to proscribe procedurality in the expression of rules, outside in the real world procedures are simply a fact of life: how else does an order progress through fulfilment, billing and collection unless through the execution of procedures? And as we seek to transfer more routine work to computer systems and give them more volition, more and more of those procedures will be executed by computer systems that use databases. Surely it cannot be inappropriate to have a computer system say “If an
order is fulfilled, raise an invoice” or “If an error occurs, send a message”?

Date’s domain/type constraints clearly have broad relevance to any type-based system. However, column, table and database constraints are variations on a single theme, dealing respectively with a single column, with more than one column in one table, and with more than one column in more than one table. These distinctions are, to coin a phrase, just geography. The number of columns or tables that a rule involves is a useful attribute of a rule in the context of the relational model, but is not a good basis for a formal rule taxonomy in a broader context because it will vary from implementation to implementation. For example, a rule that says “Every customer must have a primary contact name” might be a table constraint or a database constraint depending on how the designer decides to construct the tables.

**Object Management Group**

The Object Management Group’s “Object Analysis and Design: Reference Model” [38], says that rules are “declarations of policy or conditions that must be satisfied”, and goes on to classify rules as Constraints, which specify restrictions or requirements concerning instances of an object type, and Assertions, which are predicates that are part of the definition of an object type. (A predicate is a function that evaluates to true or false.) It attempts no more formal analysis of rules. This was recorded in 1992 in draft form, and I have been unable to trace any further discussion of the subject.

**State and State Domains**

Martin/Odell, Ross and Date have each classified rules according those properties that seem most significant to the authors themselves, but none have explored the most vital and
interesting aspect of rules: namely, that rules have the potential to express a perfect model of an enterprise, and are the first step down the path to automating such models as application software systems.

How do the editors in an enterprise (generally the executives and managers) express rules? Consider “Let’s give our good customers 1% discount on big orders over Christmas”. Good, Big and Christmas are concepts relating to the enterprise’s customers, orders and calendar respectively. In practice, all three are likely to be quite sophisticated concepts that are non-trivial to infer, and which involve multiple attributes, not all of which will necessarily be immediate attributes of the type in question.

For instance, to qualify as Good, customers might have to have an acceptable payment history of sufficient duration, and also be regularly placing orders of adequate size. “Acceptable”, “sufficient”, “regularly” and “adequate” all need to be carefully defined in turn. To qualify as Big, an order might need to exceed a certain value, which might vary seasonally, and to exclude the value of excessively high proportions of certain loss leaders or product categories that yield negative or low margins. Christmas might be defined to mean all orders received between December 1st and 15th and shipped between December 10th and 23rd.

During systems analysis, an enterprise’s editors describe to an analyst the rules that they apply, and the analyst’s job is to infer from the rules the set of types about which the database needs to store information, and what information needs to be stored about instances of each type. This vital area of communication remains fraught with misunderstanding, caused in no small measure by the existence of a layer of abstraction between the editor’s rules and the database schema that is implicit and not formally recognised. That layer of abstraction deals with what I shall call states and state domains.

State is a property of an instance that is distinct from, but derived from, its own attributes and those of associated instances. Thus the type Order might define states called Open,
Fulfilled, Invoiced, Cancelled and Archived that its instances might take. In the examples above, Good, Big and Christmas are in fact states, not attributes, of customers, orders and the calendar respectively.

Types have not just one, but multiple sets of states: as well as being open, fulfilled, cancelled and so on, an order might also be small, medium or large, or it might be domestic or export. Each set of states is called a state domain. Thus a type has multiple state domains, and each state domain contains multiple states. Here is an example of some state domains and states for the type Order:

```
Order state domain Progress
  ... state Open
  ... state Fulfilled
  ... state Invoiced
  ... state Cancelled
  ... state Archived

Order state domain Size
  ... state Small
  ... state Medium
  ... state Large

Order state domain Destination category
  ... state Domestic
  ... state Export
```

A state normally has associated with it a test which is in the form of a predicate (a function that returns either “true” or “false”), whose formal parameters are the attributes and other states of the type. The exception to this is “catchall” states that in effect have a predicate which always returns “true”.

The concept of state is intuitively familiar to most people who work in an enterprise. We readily speak of concepts such as good customers, important orders and senior managers, even though the Customer, Order and Manager types may not have attributes that might take the values Good, Important or Senior. Precisely defining states and state domains is an exacting process for editors and analysts, so once defined and estab-
lished, they are frequently re-used in different areas of a business: **Good** customers are likely to qualify for more frequent visits by salespeople, and **Big** orders are likely to be specially expedited through shipment.

It is important to maintain a sharp distinction between attributes and states: to reiterate, an instance’s state with respect to any state domain belonging to its type must be capable of being determined solely from the instance’s attributes, including its name, and by nothing else. (Note that this does not preclude a state being defined in terms of other states belonging to the type.) Also, state should not be persistent: in other words, a type should not have any attributes whose function is solely to store the value of a state domain, as this will introduce redundancy. For example, if the type **Order** has an attribute of **Size** whose value is determined by the same rules as the **Size** state domain, every time order lines were added to, amended or deleted, the state attribute would need to be re-evaluated and possibly updated. (If **Order** has an attribute of **Size** for other reasons, the predicate for the **Size** state domain should simply test the value of the **Size** attribute in order to determine an order’s state with respect to the **Size** state domain.)

**State and Database Management Systems**

Should a database management system acknowledge state? One might argue that DBMSs should concern themselves solely with recording attributes, and leave state to be managed by application code. There are two counter arguments that support the proposition that DBMSs should manage state. First, the conceptual one. A data model provides two things: abstractions to model the problem domain, and mechanisms to maintain and interrogate persistent data. As we have observed, strategies and policies are formulated and communicated largely in terms of states, not attributes, so state is undeniably part of the real world, and a data model that knowingly omits some relevant and
important aspect of the real world is an imperfect tool. If state plays a role in real life and in a data model, then it should play a role in a database management system.

The second argument is purely practical. Just like query, state is a common inference mechanism that is likely to be used by every application, and is thus more usefully and efficiently implemented by the database management system itself.

To implement state, a database management system needs two things: schema definition tools to define state domains and states for types, and an inference engine that is capable of inferring the state of an instance with respect to a specified state domain. For efficiency reasons, the inference engine should operate at as low a level in the database architecture (ie. as close to the disk) as possible.

Clearly there is common ground, and possibly common code, between state and querying. Given that the state engine extends the set of an instance’s properties by adding states to attributes, queries should be capable of being framed in terms of both attributes and states. Thus the query engine needs to be able to make requests of both the database engine itself, and to the state engine. Extending the structure, requests from the API may be addressed directly to the database engine, or to the query engine or the state engine, either of which may call on its opposite number for assistance.

Finally, it is necessary to observe for the record that state plays no part in the relational model.

Other views of State

Martin/Odell include chapters on state and state changes in [37], and define state as “a collection of associations an object has with other objects and object types”. Their concept of state includes an instance’s type (hence the phrase “and object types” in their definition) and they say that each relevant state typically represents a sub-type of the type in question. By contrast, under
the associative model, instances never change their basic type so
type does not determine state (except in the sense of furnishing
the list of eligible states) but the role performed by sub-types in
Martin and Odell’s description is similar to that performed by
subsets in the associative model.

Rational Software Corporation’s Unified Modelling Lang-

guage (UML) provides Statechart diagrams as one of its
modelling tools, requires that each object is in a particular state,
thus precluding use of multiple state domains for a type.

Both of these treatments accord state a less significant role
than the one I am advocating.

The Components of State

Naming States

The structure of a state’s name is as follows:

state ::= type specification.instance specification.state domain.state name

type specification ::= type | *

instance specification ::= instance | *

Thus a state domain may be defined for all types, or for a
specified type. We shall need to refer to states across all
instances, or as they relate to specific instances: hence the “*”.
So we can refer to:

Order.*.Progress.Open

and

Order.766564.Progress.Open

Ideally, the states within a state domain are mutually exclusive.
Thus a state of Closed would be a poor addition to the Order
Progress state domain because it would overlap Invoiced and
Cancelled. This mutual exclusivity can be difficult to achieve in
practice, and would need to be temporarily relaxed during testing, so rather than attempting to enforce mutual exclusivity, the states in a state domain are considered to be ordered, and when evaluating an instance’s state, the first state whose test returns “true” is the instance’s state. This is rather like a series of IF ... ELSE IF ... ELSE IF ... statements.

Similarly, an ideal state domain would contain every possible state in which an instance might find itself with respect to the state domain. This is also an ideal rather than a requirement, as in certain state domains only one or two states may be of interest. For example, we may wish to identify good customers without saying that the rest are in any sense “bad”. To support this, every state domain contains an implied **Undefined** state which has a null predicate and is last in order, acting as a catchall. The logic for evaluating a state is thus:

Set state to **Undefined**
Test State1; if true set state to State1 and return
Test State2; if true set state to State2 and return
Test State3; if true ...

As evidenced by the examples, my personal preference is that state domain names should be nouns, and state names should be adjectives. When a state domain contains only one or two states, it is tempting to name the domain after the states – eg. **Good** or **Bad** – but this is a bad idea. It is very likely that more states will be added, rendering the state domain name inappropriate.

**Predicates**

Nearly all states need a predicate, which a function that is capable of testing whether a given instance is or is not in the specified state. In implementation, predicates take the form of associative query request trees of the type described in Chapter 10. The instance is bound to the request, and if the instance is returned by the request, the predicate is “true”; if the instance is not returned, the predicate is “false”.

Effectual and Ineffectual States

As we seek to automate rules, as well as the capability to test states we also need the ability to set them by changing the underlying attributes to reflect the state. This process involves converting the state’s predicate to an assignment. For example, a state whose predicate is Customer status = Good may be set by changing the Customer status attribute to Good.

Some predicates cannot be converted to assignments: for example, those that involve tests for inequalities. We shall call those states that may be set effectual states and those that may not ineffectual states. For example, suppose we have:

Water temperature state domain State of matter
  ... state Solid
    ... predicate Water temperature <= 0º
  ... state Liquid not boiling
    ... predicate 0º < Water temperature < 100º
  ... state Boiling
    ... predicate Water temperature = 100º
  ... state Steam
    ... predicate Water temperature > 100º

then the only effectual state is Boiling, because it is the only one expressed as an equality. Thus an automated rules engine could set Water temperature to Boiling by setting it to 100º, but could not set it to Solid, Liquid not boiling or Steam as none of these are expressed as precise values.

When a state’s predicate contains an “or”, one side of the “or” may be effectual and the other side may not. For example, we might have a state of Premier customer whose predicate is:

    Miles flown >= 1,000,000 or Premier customer = Yes

In cases like this, a state is considered to be effectual if any part of its predicate is effectual.

It may be inconvenient for a rule whose predicate is ineffectual itself to be ineffectual, so an ineffectual rule may be rendered effectual through the addition of an “effecter”: 
Water temperature state domain State of matter
  ... state Solid
  ... predicate Water temperature <= 0°
  ... effecter Water temperature = 0°

An effector may be either a simple assignment, as above, or a series of assignments, or a stored procedure. An effector that is a stored procedure will have formal parameters drawn from the subject type, and actual parameters mapped from the instance under consideration.

Management State Domains

As well as their individual state domains, in the context of a database management system all types share a common set of state domains that deal with the management of their instances by that system. These are called “management state domains”. Here are some examples of management state domains:

• **Existence** state domain, which contains the states **Non-existent** and **Existent**. The tests for these states take an instance as a parameter, and check whether it exists in the database.

• **Name existence** state domain, which contains the states **Existent** and **Non-existent**. The tests for these states take an instance’s name and type as parameters, and check whether any instance with the same name exists in the database.

• **Validity** state domain, which contains the states **Valid**, **Invalid** and **Unchecked**.

• **Use**\(^1\) state domain, which contains the states **Used** and **Unused**. The tests for these states take a single parameter,

---

\(^1\) This is the noun, pronounced so as to rhyme with “goose”
which is the instance’s surrogate, and check whether the instance is the source or target of any association.

- **Change** state domain, which contains the states **Changed** and **Unchanged**.

**Changes**

Going back to the OED definition, a rule is a principle to which an action conforms or is required to conform. Intuitively, it seems safe to define an action as something that causes a change. (If nothing changes, what meaningful action can possibly have occurred?) Thus rules may be considered to exist in order to prohibit or to initiate changes.

In the context of a database management system, a change is that which alters, or proposes to alter, some state of an instance. There are two sorts of change: those that alter an instance from a specified state to its new state, and those that alter an instance to its new state without regard for its prior state. The syntax for a change is:

```
change ::= [state to] state
```

**Rules**

My earlier analysis of Martin/Odell’s work left two sorts of rules: structure rules and stimulus/response rules. As structure is viewed through the medium of states, a better name for structure rules is “state rules” (the name also preferred by Chris Date in [39].) State rules say either “You may not perform this action” or “You may not perform this action under these circumstances”. Here are some state rules:

- You must not walk on the grass except in an emergency
- You must not exceed 70mph
You may not cross the road when the sign says “Don’t cross”

A better name for stimulus/response rules is “change rules”, which is shorter. Change rules say “When this happens, you must do that” or “When this happens under these circumstances, you must do that”. Here are some change rules:

- When you prepare to turn left you must indicate left
- When you leave the house you must turn off the lights if they are on
- When you discover a fire you must sound the alarm

The behavioural principles that apply to the rule system are:

- You must not perform any change that results in the violation of a state rule, either directly by that change or indirectly through a change triggered by a change rule.

- You may perform any change that does not result in the violation of a state rule.

- You must perform any change that is required by a change rule (unless it would violate a state rule).

It follows from the second principle that state rules are proscriptive, not permissive.

Deadlocks may arise when a change rule mandates a change that violates a state rule. An automated rule system must be able to detect and resolve such deadlocks.

The BNF form of a state rule is:

\[
\text{state rule ::= change prohibited [when [not} \text{ state \{and|or [not] state\}] always]} \]

and of a change rule:

\[
\text{change rule ::= change mandates change \{; change\} [when [not] state \{and|or [not] state\}] always} \]

Also, to reiterate:

\[
\text{change ::= [state to] state} \]
Let’s try some examples. We’ll start big: “You shall not kill”, which we can express through a single state rule:

\[
\text{Person state domain State of being} \\
\quad \text{... state Alive} \\
\quad \text{... state Dead by natural means} \\
\quad \text{... state Dead by unnatural, accidental means} \\
\quad \text{... state Dead by unnatural, deliberate means} \\
\text{Person.*.State of being.Dead by unnatural, deliberate means} \\
\text{prohibited always}
\]

Let’s try some rules involving instances: a formal declarative assertion of “Mary will marry Bill when Hell freezes”. We’ll absolve them both from bigamy or worse by ensuring that they’re both alive and unmarried in the event of this unlikely occurrence:

\[
\text{Person state domain Marital status} \\
\quad \text{... state Dead} \\
\quad \text{... state Married to Mary} \\
\quad \text{... state Married to Bill} \\
\quad \text{... state Unmarried} \\
\text{Afterlife state domain Temperature} \\
\quad \text{... state Frozen} \\
\quad \text{... state Not frozen} \\
\text{Afterlife.Hell.Temperature.Frozen} \\
\text{mandates} \quad \text{Person.Mary.Marital status.Married to Bill; Person.Bill.Marital status.Married to Mary} \\
\text{when} \quad \text{Person.Mary.Marital status.Unmarried} \\
\text{and} \quad \text{Person.Bill.Marital status.Unmarried}
\]

**Rules in a Database Management System**

Let us now consider a framework for rules in the context of a database management system. First, we need a firm platform from which to operate. (Rules are slippery things that can rapidly retreat beyond our reach – it would be inappropriate for example to have to attach conditions such as “Provided the world still exists” or “Provided the computer is switched on” to
every rule.) So we shall take as our platform the existence of a
correctly-operating database management system, with a well-
formed structural schema that defines:

- a set of types
- their attributes, including cardinalities
- their state domains and states, with predicates (we will
  consider how predicates are expressed later)
- an appropriate set of datatypes that encapsulate their own
  validation methods.

- a set of management state domains

This means that we shall take the platform as a given, and that
we will concern ourselves solely with rules that are expressed
above the platform. This is not to say that there are no rules
below the platform: we could choose to express the structure of
the schema as rules, but this is rather like working on a car’s
engine at the same time as you are driving it, which is tough to
do and not recommended.

We need in addition two more processing engines:

- a state engine that is able to ascertain the states of instances
  in the database both before and after transactions are applied,
  and is capable of initiating changes of state, and

- a rule engine that is able to determine all the rules that are
  triggered by a transaction, ascertain whether they will be
  breached were the transaction to proceed, and determine
  what changes would be required as a consequence of the
  transaction were it to proceed.
With this platform, and using the definitions that we established above, we can make a start on a framework for automating rules. We’ll begin with some global rules affecting all instances of all types. The first says that we cannot create a new instance of a type unless it is valid:

\[ *.*.Existence.Non-existent \text{ to } *.*.Existence.Existent \text{ prohibited when } *.*.Validity.Invalid \]

The second says that we cannot delete an instance that is in use:

\[ *.*.Use.Used \text{ to } *.*.Existence.Non-existent \text{ prohibited always} \]

It may of course be sensible to decide that such fundamental and broadly-applicable rules should be applied by the database engine at a lower level than the rule engine: however a viable framework for rules must be capable of expressing all rules that operate above the platform, regardless of how they are implemented.

Now let’s formulate some rules about the state domain of Order that we were considering above:

Order state domain Progress
... state Open
... state Fulfilled
... state Invoiced
... state Cancelled
... state Archived

Order.*.Existence.Non-existent to Order.*.Existence.Existent
prohibited when not Order.*.Progress.Open
Order.*.Progress.Open to Order.*.Progress.Invoiced prohibited always
Order.*.Progress.Open to Order.*.Progress.Archived prohibited always
Order.*.Progress.Fulfilled to Order.*.Progress.Cancelled prohibited always
Order.*.Progress.Fulfilled to Order.*.Progress.Archived prohibited always
Order.*.Progress.Invoiced to Order.*.Progress.Cancelled prohibited always
Order.*.Existence.Existent to Order.*.Existence.Nonexistent prohibited always
Order.*.Progress.Fulfilled mandates Order.*.Progress.Invoiced
Order.*.Progress.Invoiced effector Order.*.Progress.Invoiced()

Let us see exactly what the various components in the diagram above have to do.

1. The API receives a transaction, in the form of a request to create instances, and/or to delete instances, and/or to amend certain attributes of certain instances to new values.

2. The API invokes the state engine to ascertain the current state of each affected instance, and the state into which it would be placed were the transaction to proceed. New instances are in the Nonexistent state of the Existence management state domain.

3. From this information the state engine infers any changes that are mandated by change rules as a result of these changes, and the mandated changes are added to the transaction.

4. Steps 2 and 3 are repeated recursively until no further changes are added to the transaction.

5. For every change in the transaction, the API invokes the rule engine to determine whether the transaction would violate any state rules. If it would, the transaction is rejected and the sequence ends.

6. The API invokes the state engine to translate changes expressed as effectual states into database updates.

7. The transaction is applied.
12. BACKGROUND TO THE ASSOCIATIVE MODEL

The associative model has elements in common with models and techniques variously called binary relational, entity-relationship and triple store. The first two are conceptual models with a number of features in common and the third is an implementation technique. I shall use the term “binary model” to mean an abstraction of the binary relational and the entity-relationship models. (In the literature, both binary relational and entity-relationship come with or without a hyphen, and with or without the final “...al” or “...ship”.)

Historical Perspective

Chen’s 1976 paper “The Entity-Relationship Model – Towards a Unified View of Data” [23] is widely credited as the origin of the binary model, but earlier work, notably by Feldman in 1965 [16], Levien and Maron in 1967 [24], Ash and Sibley in 1968 [25], Feldman and Rovner in 1969 [26], and Titman in 1974 [27], belies this. In 1976, at about the same time as Chen, Bracchi, Paolini and Pelagatti in Milan proposed binary logical associations as an appropriate basis for data modelling in “Binary Logical Associations in Data Modelling” [28].

binary model using triples. He went on to describe more of his experience in his 1982 paper “Binary-Relational Storage Structures” [20]. In 1992, John Mariani at Lancaster University built and described “Oggetto: An Object Oriented Database Layered on a Triple Store” [32]. Today, the Tristarp (short for “Triple Store Application Research Project”) group under Peter King at Birkbeck College, London has built and commercially deployed a triple store database [33].

Lastly, some of the work done on semantic networks touches common ground with the binary model, most notably “Using Semantic Networks for Data Base Management” [34] by Nicholas Roussopoulous and John Mylopoulos at University of Toronto, published in 1975.

The binary model has been extensively researched, and several authors have commented on its perceived shortcomings. Some of these comments are as relevant to the associative model as they are to the binary model, and some are not. Naturally enough, much of the criticism has come from the staunchest advocates of the relational model. Codd mounts a comprehensive attack in his book, whilst Chris Date and Hugh Darwen in [10] simply take it as axiomatic that we want to stay in the relational framework. To them, “it would be unthinkable to walk away from so many years of solid relational research and development”. No doubt typewriter manufacturers said something similar.

The Associative Model versus the Binary Model

The associative model differs conceptually from the binary model and in implementation from triple stores in one fundamental way: associations themselves may be either the source or the target of other associations.
Thus in the diagram, A1, A2, A3 and A4 are all valid and different shapes of associations between the entities E1, E2, E3 and E4 and each other.

- A1 associates the two entities E1 and E2
- A2 associates the association A1 with the entity E3
- A3 associates the entity E4 with the association A2
- A4 associates the association A3 with the association A1

This capability is not found either in the binary model or in triple store implementations.

**Associative Model versus Binary Model**

Recapping briefly, the associative model classifies things in the real world as either entities, that have discrete, independent existence, or associations, whose existence depends on one or more other things, such that if any of those things ceases to exist, then the association itself ceases to exist or becomes meaningless.

Under the binary model, an entity’s properties are recorded by means of its associations with other entities, but these
associations are not recognised as things in the real world in the same domain as entities. Thus associations under the binary model may not themselves in turn have properties of their own. The types that associations instantiate have attributes such as name and cardinality, but these are meta-properties: that is, they are properties of types of associations that are used in the context of the model, not properties of instances of association type that are used in the context of the real world.

Under the associative model, entities and associations are peers: each represents things that occur in the real world, and each may have properties that are represented by associations. These associations may themselves have their properties in turn, that are similarly represented by associations.

As the diagrams below show, under the binary model entity types and association types are the most abstract constructs that are employed. Under the associative model, entity types and association types are both subtypes of the more abstract notion of a type. A type represents a group of things in the real world, whose members are either entities or associations but not a mixture of both.

This capability of the associative model to record the properties of an association adds significantly to its expressive power, and allows it to model the real world more accurately than the binary model.
Chapter 12. Background to the Associative Model

Binary Model

Associative Model
The Associative Model of Data

Model level:

Legal entity — sells — Product

Real world:

Safeway — sells — Cornflakes 500gm

associates

Binary Model

Model level:

Legal entity — sells — Product — Price

Real world:

Safeway — sells — Cornflakes 500gm — £1.23

associates

Associative Model
Associative Model versus Triple Store

As I mentioned above, triple store is an implementation technique that has been described in conjunction with the binary model. The term “triple” means a 3-tuple, each of whose columns contains a reference of some type to an object stored in an accompanying name table. A “triple store” is a table of such triples together with its name table. Here is an example of a triple store that stores two tuples:

Mary Jones lives in Britain
Amazon sells Dr No

<table>
<thead>
<tr>
<th>Name table</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>41</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>67 90 14</td>
</tr>
<tr>
<td>23 38 41</td>
</tr>
</tbody>
</table>

Under the associative model, the 3-tuples in essence become 4-tuples, where the new first column (shaded in the example) is the identity of the tuple itself:

Mary Jones lives in Britain
Amazon sells Dr No
... for £9.50
<table>
<thead>
<tr>
<th>Name table</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>41</td>
</tr>
<tr>
<td>53</td>
</tr>
<tr>
<td>98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Associations</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
</tr>
<tr>
<td>77</td>
</tr>
<tr>
<td>03</td>
</tr>
</tbody>
</table>
13. CHALLENGES OF THE ASSOCIATIVE MODEL

The associative model will seem challenging to database practitioners schooled in the relational model. This chapter discusses some of its more challenging aspects.

Abandoning the Record

The associative model does not use records. From punched cards through to the object and object/relational models, the basic unit of data storage has been a record that comprises all of the individual pieces of information about an object or an entity, stored contiguously. The chief argument in favour of the record has been efficiency: given that visiting the disk is a slow, mechanical process, the more data that can be retrieved during each visit the better.

Efficiency has been at the forefront of concerns about the binary model, and hence the associative model also, because both models abandon the record-based approach used by all the other data models in favour of storing data items individually. But as the power of hardware continues to increase, absolute efficiency is progressively sacrificed to gain other benefits, as happened in the evolution of programming languages from machine code through assembler to third and fourth generation languages. In this light, the benefits of adopting a more granular approach to data storage and retrieval – that is, storing data in smaller units – should now be considered.

A record comprises all of an entity’s data items, stored contiguously. The concept of the record originates with and is best exemplified by the punched card. On a card columns 1 through 20 might have held the customer’s name, columns 21 through 30 their outstanding balance, 31 through 40 their credit limit and so on. The record is an explicit feature of the
hierarchical and network data models, and closely corresponds to the tuple in the relational model. Abandoning the record is rather like cutting up each punched card into vertical sections, 1 through 20, 21 through 30 and so on, and maintaining an index of where to find each section. This means that an entity’s data items are no longer necessarily stored contiguously (either conceptually or physically) and so to retrieve all of them usually requires more than one visit to the disk, whereas a record comprising all of an entity’s data items can usually be retrieved in a single visit, as a punched card could be read in one operation.

To this extent non-record-based models (I shall call them granular models) are inherently less efficient than record-based models, including the relational model. However, the margin of difference is not so great as might be thought.

In a well-normalised relational database, most relations contain a fairly high proportion of foreign keys – in certain types of complex transactions such as sales orders, it is not unusual to find foreign keys in more than half the columns. Working interactively, good user interface design dictates that some meaningful data is presented from each tuple whose primary key appears as a foreign key, so that the user can have visual confirmation that the application has got it right. For example, if customers are identified by account numbers, and an order carries an account number as a foreign key, it would be usual to present the customer’s name alongside the account number. Similarly, working in batch mode, it is often necessary to retrieve the tuples identified by foreign keys in order to get the full picture about an entity: in extending a sales order to create an invoice, prices, product descriptions, discount rates, sales tax rates and so must all be retrieved by means of foreign keys. The bottom line is that at least one additional tuple is likely to be retrieved for every foreign key.

In a modern, well-normalised sales order processing application, it is not unusual to find that tuples must be retrieved
from a dozen or more different relations in order to present a single sales order on the screen. Suppose that such an order comprises one header tuple with twenty columns, plus ten detail line tuples each with eight columns, where half of the columns in each relation are foreign keys. Under the relational model, the number of tuples that need to be retrieved to assemble the whole order is not the number of tuples in the order – 11 – but this number plus one for each of the 50 foreign keys, giving a total of 61. Under the granular model the number of items and links to be retrieved approximates to (depending on the exact design) the original number of columns – 100 – plus one for the target of each column, giving 200 in total.

So although in practice granular models are indeed less efficient in minimising disk accesses than record-based ones, the margin of difference is not nearly so great as it might appear to be: in this case, just over three to one. Anyone who uses the relational model has already accepted a substantial trade-off in efficiency; if minimising disk access was the sole consideration, sales orders would be stored in un-normalised form. Each could then be retrieved in a single visit to the disk, yielding a margin of efficiency over the relational model of more than sixty to one.

Most software innovators agree that it is important not to under-estimate by how much the power of hardware will increase during the lifetime of their product, and consequently how the trade-off between functionality and performance will alter. In terms solely of the amount of work that a computer has to do to present a screen-full of information to a user, the relational model is more efficient than the associative model. But the same can be said of second generation programming languages compared to third generation. As computer power becomes ever cheaper, the right question to ask is not “Is A more efficient than B?”, but rather “How much benefit does B offer in return for the cost of some of A’s efficiency, and is the trade worth it?”. From this more enlightened standpoint, the associative model wins.
Distinguishing Entities and Associations

The associative model divides things into two sorts: entities and associations. Entities are things that have discrete, independent existence, whilst associations are things whose existence depends on one or more other things. Previous data models have made no useful distinction between the two, or, to be more precise, have demanded that associations be modelled as entities if their properties are to be recorded. The associative model acknowledges the distinction as one that occurs in the real world, and thus one that allows the creation of more accurate models of the real world. As we discussed in Chapter 5, a series of benefits flow from this.

One of Codd’s principal objections to the binary model is that, in his words, one person’s entity is another person’s relationship, and there is no general and precisely defined distinction between the two concepts. I disagree. Firstly I believe that most sensible people, once the distinction is pointed out to them, are readily able to decide whether something is an entity or an association. Secondly it is possible to define the distinction between entities and associations in a simple, reasonably intuitive but nevertheless rigorous way. Such design decisions are almost trivial compared to some of the decisions a relational practitioner is called upon to make when designing base relations.

Codd also objects to the entity-relationship model on the grounds that it does not allow associations to have properties. He is quite right to do so, and the associative model rectifies this, without requiring that they be modelled as entities.

However, much of Codd’s ammunition is wasted because the version of the binary model at which he aims his criticisms is not the one contemplated by most researchers. Codd assumes that there is one distinct two-column table per entity per association type, and the two columns of each table are the two associated entities. In fact, most interpretations of the binary
model assume that the association type forms the third column, and as we shall see, when this is the case a relational schema comprising any number of relations can be represented in the binary model by just two relations in total: one for entities, one for associations.

**Naming Entities**

Under the associative model, every instance of an entity type requires a name. There is no parallel concept in the relational world, and some people take issue with the requirement for naming, preferring that a thing’s name, when it needs one, should simply be one of its attributes.

Naming things is a vitally important feature of human discourse, so it is irrational to dispense with it when we move into the world of databases. In the associative model, the entity name concept avoids the need to choose which attributes will be used to identify something visually (remembering that every entity has its own unique surrogate anyway, so identity is assured for internal purposes) or to create redundant identifiers such as country codes and customer numbers.

Three arguments predominate. First, that not everyone necessarily refers to something by the same name, so it is artificial to require entities to have one single, special name. This is easily resolved: simply create an association `Person` has alias `Name` with multiple cardinality, and set `Name` as a supertype of `Person`. Then `Name` will contain all instances of `Person` and all instances of `Name`, and any entity can have as many aliases as desired.

The second argument is that a name, when needed, is best derived from an entity’s attributes. The example invariably cited in this context is that of the entity type Person, which leaps most readily to mind, but is also probably the least typical in this respect. The argument runs that we are likely to be storing a
person’s first name and family name, so surely we should derive a name as needed by concatenating the two, whichever way round we prefer, and that to store a separate entity name is redundant.

This doesn’t stand up to close analysis, ignoring as it does the distinction between formal and everyday names. When we store a person’s first name and family name, these must normally be the full and formal renditions – Joseph, not Joe, for example. So if we had an individual with attributes of first name “Joseph” and family name “Williams”, this would still not help us identify the person that everyone knows as “Joe Williams”. The same is also often true of companies: in our records, we would need to record “International Business Machines Corporation” for contractual purposes, but you can bet the folks in purchasing refer to them as IBM.

The third argument is that it is difficult and inconvenient to assign names to entities of certain types: meetings, purchase orders and so on. Under the associative model, this is a sure sign that these things should be represented as associations, not entities: in the case of a meeting, perhaps between a venue and a date; in the case of a purchase order, perhaps between a supplier and a timestamp. In other words, if something’s name is not readily evident, it’s probably an association.

**Using References, Not Values**

The associative model recognises scalar values and strings as things in their own right, with independent existence and identity, instead of as isolated values that represent objects. This approach substantially reduces the amount of work needed to execute queries, and has other benefits – if today’s databases had incorporated this capability, the Millennium bug would have been resolved with a fraction of the resources actually consumed. However, the approach depends at a fundamental
level on the use of references or pointers to values instead of values themselves, and this is contentious.

Both Codd and Date have issued stern injunctions against the use of pointers in the relational model. Date has taken the argument furthest in his book “Relational Database Writings 1994 – 1997” [35], which contains two chapters on pointers and a third on object identifiers, and he describes the introduction of pointers into relations as the Second Great Blunder.\(^1\)

The question at the heart of the issue is whether pieces of data should be represented in a database solely by values, in accordance with Codd’s information feature for the relational model, or by references to variables that contains values, in accordance with the associative model, or either, at the user’s election.

There is also a secondary question of whether things whose properties are recorded in a database should be identified by keys or by surrogate keys. A key is some unique combination of a thing’s existing properties, whilst a surrogate key is a new property assigned as the thing enters the database, solely for the purpose of identifying it and for no other purpose.

Surrogate keys look like object identifiers (as they are commonly used in the object model) in many respects, but Date makes a distinction between surrogate keys and object identifiers and rightly concludes that, whilst object identifiers perform some of the same functions as surrogate keys, they carry a lot of additional baggage with them, and thus are not the same thing as pointers.

In [6], Codd excludes pointers from the relational model because he believes that both programmers and end-users find them difficult to understand. He cautions us that “the manipulation of pointers is more bug-prone than is the act of comparing values, even if the user happens to understand the

\(^1\) The first Great Blunder, in Date’s view, was the perception that relations – or, to be precise, the intensions of relations, or relvars as Date calls them – are equivalent to object-oriented classes.
complexities of pointers.” However, Codd also makes it clear that his prohibition extends only to pointers that are visible to users: “It is a basic rule in relational databases that there should be no pointers at all in the user’s or programmer’s perception.” (My italics.) He goes on to concede that “For implementation purposes, however, pointers can be used in a relational database management system ‘under the covers’, which may in some cases allow the DBMS vendor to offer improved performance.”

In [10], Date and Darwen issue a specific proscription, namely “No value shall possess any kind of ID (identifier) that is somehow distinct from the value per se”, and consequently reject the notions that other objects might make use of such IDs to share values and that users might have to de-reference such IDs, either explicitly or implicitly, in order to obtain values.” (“De-reference” means to retrieve whatever it is that a pointer points to.)

Regarding the use of pointers in the relational model, I agree with Codd and Date, with the exception of a single caveat which I shall describe in a moment. The relational model has no need of visible pointers to achieve its goals and was explicitly designed to dispense with them. Moreover the relational model relies extensively on the use of predicate logic to compare values directly, and this function is undermined and rendered more complex by the use of pointers. Certainly you can add pointers to the relational model, but to do so would be a significant departure from the relational model, and the clear and sturdy conceptual basis of the relational model would be degraded. If the modification adds value without undesirable side-effects, well and good. However in this case the case for the added value is not clearly made and the side-effects have not been explored. At some point the custodians of a conceptual model must defend it from further degradation.

Now for the caveat. The relational model’s use of primary and foreign keys has sufficient similarities to a pointer mechanism (albeit one entirely exposed to the user) to cause me
to wonder whether Codd and Date protest too much. Moreover, as a pointer mechanism it is fragile: unless the prohibition of duplicate tuples in relations is rigorously enforced, which it is not in many commercial implementations of the relational model, one cannot guarantee always to be able to unambiguously de-reference a foreign key.

Date’s aversion to pointers does not extend to surrogate keys. In the context of the relational model, a surrogate key is a key like any other and identifies a single row, but it is not composite, it serves no other purpose and is never reused, even after the thing that it identifies is removed from the database. In [37] he says “Surrogate keys are a good idea (frequently, if not invariably ... ). More specifically surrogate keys can help avoid many of the problems that occur with ordinary undisciplined user keys.” So, the associative model’s use of surrogate keys that are invisible to both the programmer and the user, and are not object identifiers, does not of itself violate the principles that Codd and Date have articulated.

(Date doesn’t say explicitly whether a row with a surrogate key would be identified within a database solely by its surrogate key, or by the name of its relation together with surrogate key. He perhaps implies the former by saying that surrogate keys would never be reused, but this further implies that there must be a way to infer from a surrogate key the name of the relation in which it can be found.)

Where the associative model is most fundamentally at variance with the relational model is in the second question: should data be represented by values, or pointers to variables, or either? The relational model, in accordance with Codd’s information feature, does only the former. The associative model does only the latter. There are two cases to consider: where the database is representing relationships between one entity and another (which the relational model implements using foreign keys) and where the database is storing a scalar value or a string.
Before you pass judgement, I shall examine the associative model’s behaviour more closely.

Within any reasonable problem domain, the integer 12, the monetary value $12.00 or the string “QWERTY” all have unequivocal identity. They also qualify as entities according to our test: there is nothing in the real world which, if it ceased to exist immediately, would render the thing in question non-existent or meaningless. They also each have an obvious identifier, which is their own value.

Most modelling systems and programming languages (except Smalltalk) do not treat scalars and strings as objects or entities: instead they use a value that represents the object. But there is a crucial difference between the entity that represents the decimal integer 100, and the different values that may also be used to represent it, such as 100, or 100.00, or 000000100.000000, or 1.00E+002. To illustrate the point, we simply have to alter the number system that we are using from decimal to hexadecimal, and the values then refer to a different integer entirely.

Suppose we are building a database that stores addresses. If we put the string “London” in several different columns of several different relations, each time we enter the string again we create an entirely new representation of it, and the database makes no attempt to see if it has already stored the string “London” before, or to try to re-use it. So we may end up with the string “London” stored, say, 1,000 times in the database.

There is nothing to say whether all these values refer to one town, or to more than one – our database may refer to any number of towns called London between 1 and 1,000. If one of these Londons were to change its name, first we would have to locate each one, and then decide whether it was the one which had changed its name or not.

The mechanism that the relational model provides to address this is to allow us to create a relation called Towns, and within it a tuple for each different London. The primary key of
to each tuple can then be used as a foreign key in various tuples of other relations to refer back to the appropriate London. However, as the issue arises every time for every scalar and every string, it is fair to say that whilst the relational model does not prohibit this approach, if it had wished to endorse it, it would have made it much simpler to implement. Thus in practice if not in theory, it prohibits it.

These observations are equally relevant when we are dealing with, say an amount of money or a date; however there is usually less scope for ambiguity with scalar values. “01-Jan-2000” or “$100” are pretty unambiguous whether they occur as the identity of instances or as values. But there is still a world of difference between a value that represents an instance and the instance itself. If our database had stored identities of dates instead of dates as values, the Millennium bug would have had a fraction of the impact that it is currently having.

Moving Away From Object Orientation

The associative model is intentionally not object oriented and is not compatible with the object model of data. Object orientation is a powerful and important programming technique. But the guiding principle behind its invention was to restrict or prohibit access to data in main memory in order to ensure its integrity. In fact, to borrow Date’s words from [37], “The ‘object model’ is a storage model, not a data model.” Date puts the phrase “object model” in quotes because, as he points out, there is no universally agreed, abstract, formally defined “object model”. This is simply not an adequate starting point for tools whose primary function is to provide, in Codd’s elegantly simple phrase, shared access to large data banks.

It should not be inferred from this that the associative model is not compatible with object-oriented programming languages: nothing could be further from the truth. To use an object-
oriented programming language in conjunction with a database based on the associative model (or, indeed, on the relational model) is simply to acknowledge that relatively small amounts of transient data in a computer’s memory should not necessarily be organised, managed or protected in the same way as significantly larger volumes of persistent data in a shared database.

Our own implementation of the associative model is written in Java, and its APIs are delivered as Java packages.

**Re-asserting the Nature of the Problem Domain**

The associative model reasserts the nature of the problem domain that database management systems should be addressing. Object oriented database technology has failed to find a commercially sustainable market either as a repository for multimedia files or as persistent storage for object-oriented programming languages.

The opportunity for the next generation of database management systems lies not with objects or universal servers (except to the extent that they provide a richer set of datatypes for the relational model), but in using vastly increased hardware resources to improve on the way that we store and query our core mission-critical enterprise and transactional data, on which the financial and sometimes physical well-being of enterprises and individuals depends.

**Fewer Programmers?**

Often during discussions of the associative model and omni-competent programming, the question comes up as to whether their widespread adoption, were it to come about, would mean that fewer programmers would be needed. Speaking as someone
who, in 1969, was wondering whether my job as an Easycoder assembler language programmer would survive my employer’s purchase of a COBOL compiler for our Honeywell H200, I understand the concern, but it is unfounded.

Omnicompetent programming certainly means that less procedural code is needed. But procedural code is the chaff that occupies a lot of bulk, and frequently obscures the kernels of the business rules that, together with data structures, are the real intellectual property in database applications. If more programmers are freed from repetitious coding, they will be able instead to add more value for their employers by developing more applications more quickly.

In any event, conventional wisdom now has it that around 80% of our programming resources go towards maintenance, and 20% towards new development. But I’d be willing to bet that the ratio of programmers who prefer maintenance to those who prefer new development would be pretty much reversed, with 80% or more opting for new development. So programmers should be the first to embrace the associative model.
14. THE RELATIONAL MODEL REVISITED

I have asserted and discussed elsewhere that there are six significant limitations of the relational model that open the door for an alternative. This chapter discusses some of the other issues that should be taken into account if the relational and associative models come to be discussed as peers.

Conceptual Foundation

Today, the relational model is a moving target. Codd himself has described three versions of the model – RM/V1, RM/T and RM/V2. Each relational vendor has its own version, all of which deviate from Codd’s definitions in various ways, by failing to implement certain features and by implementing others – often compromising the relational model – that the vendors judge to be of value to their own group of customers. Date and Darwen have consolidated and refined the relational model in their object/relational model, whilst still staying largely true to Codd’s original vision. Most of the features of Codd’s RM/T and RM/V2 have not been generally implemented and are now unlikely to be, and the market has yet to pass judgement on the object/relational model.

A significant body of literature has grown up around the relational model. This would be a good thing if it were not for the gulf that has developed between theory and practice, but most of the commercial implementations of the relational model are not particularly true to its theoretical foundation. One senses that the theorists are shouting into the wind whilst the vendors pursue their own agendas, which, reasonably enough, focus mainly on satisfying their customers, whose own priorities have more to do with cost and efficiency than conceptual purity. The result is confusing for theorists and practitioners alike.
Relationships and Identity

In the relational model, relationships are implicit, not explicit. Notwithstanding its name, the relational model does not explicitly record relationships between entities. Relationships are inferred when the values of the primary key columns of one tuple are matched in the corresponding foreign key columns of another. However, if there is more than one tuple in the target relation with identical primary keys, it is possible that a referenced entity cannot be uniquely identified.

But surely tuples and primary keys are always unique? Codd insists as a fundamental tenet of the relational model that no relation should be permitted to contain two identical tuples. This is well and good, but many implementations of the relational model do in fact permit duplicate tuples, as indeed does the model itself from a purely structural perspective. Given that the need to avoid duplicate tuples is so fundamentally important, the absence within the model itself of a mechanism to ensure that duplicate can never be created must be considered a weakness.

Moreover, an entity in a relational database has no identity except the primary key of the tuple that records its properties. If the primary key is changed, the entity loses its identity and loses touch with any other entities that were associated with it, including entities that may have been part of it, such as sales order detail lines. There is no mechanism within the relational model to detect all foreign keys in all relations that contain the value of the primary key being changed, so if a key is changed unintentionally, information will be lost.

In a relational database application, those properties of an entity that are represented by associations with tuples in other relations (as opposed to those that are represented by values in columns of the same relation) are generally the more important ones, because both their source and their target are entities that have properties of their own, and such entities are usually of
more significance to the user that those entities that are represented simply by values in columns. For example, it is more important to know that a particular sales order was placed by Customer X than to know that it was placed on Date Y (not least because if one knows the former, the latter may be obtained).

All in all, one can remark with some justification that the relational model’s key mechanism is a relatively fragile way to ensure the integrity of a database and to keep track of the more important facts within it, dependent as it is on the vendor’s resolve to prohibit duplicate primary keys and to control and manage changes to primary key columns.

**Design Complexity**

Thirty years after the inception of the relational model, designing relational database applications is still a highly specialised skill that demands considerable know-how and years of experience and insight. The design process has so many variables that no two designers are likely ever to produce the same solution to a given problem. This undesirable state of affairs has arisen because the relational model allows considerable leeway and offers little guidance in three critical areas: choosing which relations to implement, choosing which columns each relation shall have, and choosing which columns shall form the primary keys of each relation.

**Implementation Independence Compromised**

One of the avowed objectives of the relational model is to create an implementation-independent layer to isolate programmers from implementation considerations. However, in practice, implementation considerations often intrude into the design
process at a fundamental level. They come from two sources: relational schemas that are less normalised than they should be, and the use of triggers and stored procedures.

Sub-optimal Normalisation

There is a common perception that a fully-normalised database schema sacrifices performance for the sake of other less immediate and less tangible benefits, and so many relational database applications are less than fully normalised by design. Such design trade-offs typically optimise performance in favour of the design objective that was most immediate at the start of the design process, which is rather like going away for a year’s holiday taking only those clothes that are appropriate for the season in which the holiday starts.

But change is the only constant, and for any new application one can be certain that the goal posts will move, and quickly. Indeed the changes in working practice brought about by the deployment of a new application is often the first factor that impacts it, just as the volume of traffic attracted onto a new freeway often invalidates the assumptions that governed its design.

The relational model offers the freedom to model the real world in a sub-optimal way for short-term benefit, and this is a weakness.

Triggers and Stored Procedures

Most implementations of SQL (although not the relational model itself) support the enforcement of rules about data through the use of stored procedures whose execution is initiated by events called triggers. A trigger is deemed to be fired before or after either an individual update or a connected set of updates to a database. This mechanism is flawed for several reasons.
Firstly, the use of stored procedures means that application logic is fragmented: some of it lives within the application programs themselves, and some of it lives within the database. Thus there is a greater likelihood of the left hand not knowing what the right hand is doing.

Secondly, each major relational vendor has chosen a different language in which to write these procedures, adding to the relational Tower of Babel. Thirdly, the level of interdependence in large schema is such that data can be placed in breach of a constraint as a result of an apparently unconnected update: under such circumstances, unless the scope of a stored procedure ranges very widely indeed (which exacerbates the first issue), triggers are of limited use. Lastly, the use of stored procedures undermines one of the fundamental premises of database: that data shall be a resource available independently of programs.

**Managing Historic and Future Values**

Most companies who change their address send out advance notification specifying the date from which the new address should be used. Most relational applications would be unable to deal with this: there is no natural mechanism in the relational model to deal with old or future values, and few designers bother to build in such mechanisms for non-critical pieces of information. But every piece of information in a database is susceptible to change. A good data model should not require designers to go out of their way to allow changes to be recorded ahead of their effective time, or to allow users to see not only the current value, but the previous and, where relevant, the next values, of a piece of information.
Values and Domains

In the relational model, values must always be atomic: that is, not decomposable into smaller pieces. Values that are not atomic are called compound, and the only type of compound data that the relational model permits is the relation itself. However, there are many instances of quite simple values that are in fact compound, but whose existence as a relation would be overkill. An amount of money comprises a decimal value plus a currency symbol. A date comprises a year, month and day. (SQL supports the DATE, TIME, TIMESTAMP and INTERVAL data types, but these clearly violate the atomic data rule.) A UK postcode comprises two parts, each of which have meaning.

Also, the insistence on atomic data foregoes an opportunity to add meaning. There is no way in a relational database to indicate that a forename, middle initial and family name together comprise someone’s name, or that two lines, city, state and zip code comprise an address.

The values in any column of a relation must all be drawn from the same domain. Whilst in most cases this is appropriate, in some cases it isn’t. Take postal codes: the U.S. zip code is an integer in the range 10000 to 99999; the UK postcode is alphanumeric in the form “AA9X 9AA”. If domain rules were rigorously enforced, a relation containing international addresses would require a separate column for each different type of postal code. This rule also causes problems around the subject of nulls – the domain rules have to be relaxed to permit the insertion of the null mark, but this is a slippery slope. Codd defines two types of null mark; one for “unknown” and one for “unknowable”, so why not a third for “currently unknown but steps are in hand to determine”, a fourth for “known but not in a valid form” and so on?
Modelling Cardinality

In the real world, association types have a property called cardinality that says how many instances of the target type may be associated with each instance of the source type, and vice versa.

For example, in a billing application, we need to assert how many despatches an invoice may cover, and on how many invoices parts of one despatch may appear. The answer in each case is one of four possible values: zero or one, exactly one, one or more, or any number (including zero), depending on how we operate our business. These four values are often written 0,1; 1,1; 1,M and 0,M respectively.

When associating despatches with invoices, a cardinality of 0,M:1,1 would mean that each despatch must appear on exactly one invoice, and each invoice may cover any number of despatches, including zero. Each side may take one of the four values, giving sixteen possible cardinality values for one bi-directional association type.

In the relational model, cardinality cannot be fully modelled or enforced. Of the sixteen possible values, only two can be readily modelled: 0,M:0,1 and 0,M:1,1. The other fourteen cardinality values all require some combination of extra relations, columns or reference counts.

Special Values

Nulls

In a relation, the intersection of every tuple and every column (which I shall call a “cell”) exists, conceptually if not physically. Where the contents of a cell is unknown or unknowable, the relation model requires that a special mark called a “null” is
inserted into the cell. In RM/V2 Codd defines two types of such marks: “A-marks” for information that is missing but applicable, and “I-marks” for information that is missing but inapplicable.

Where a column is sparsely populated, as for example the fourth line of an address, this represents an overhead. Also, the concept itself is ill-framed: Codd has chosen two particular states of a cell that do not represent actual values, but there are many more: information may be “applicable and known but not valid”; it may be “missing and applicable but currently being sought”, and so on. The debate about nulls has generated more heat and less light than almost any other feature in the relational model, and it all stems from the conceptual flaw that allows every cell to exist (conceptually, if not physically).

**Default Values**

If we allocate a credit status to each of our customers, which may be either “Good” or “Not good”, and 98% of our customers are “Good”, rather than recording the value “Good” against the 98% of customers to whom it applies, it is sensible to have a mechanism that says, in effect, “If the credit status is not recorded, it’s ‘Good’”. “Good” in this case is called a default value.

The relational model has no concept of a default value, so it is necessary to record a value for credit status against every customer, when in reality it would suffice to record a value only for the 2% of customers who are “Not good”. This is rather like a tailor having to record that each of his customers has two arms against the day that he may gain a one-armed customer.

**Type Values**

Type values are values that apply to every instance of a type, rather like the object-oriented concept of a class variable. Suppose that we have a relation for cars: the only way to record
the fact that every car has four wheels without creating another relation is to create a column for “number of wheels” and record the value “4” on every tuple.

**Web Services**

The shift to web services, if it transpires to the extent being predicted at the time of writing, will have a fundamental impact on the nature of applications. Business functions will be automated not by monolithic local applications, but through a number of individual application services from a wide range of service providers, deployed and accessed via the web. This in turn has far-reaching implications for database architecture. Relational databases are ill-suited to a web services environment, for three reasons:

- Web services require a unified, real-time view of a data universe that selectively aggregates schema and data from a number of individual databases, running in different locations and platforms. This presents massive logistical challenges for relational technology.

- Relational technology lacks concepts such as supertypes and equivalence that are needed to aggregate schema and data from separate databases into a coherent view.

- Relational technology has no concept of absolute identity within a data universe. This works in a monolithic environment, but not in a federated one where each member is likely to have chosen different attributes as primary keys.
15. BENEFITS OF THE ASSOCIATIVE MODEL

Many people would argue that the relational model has served us well for many years, and, with some enhancements to incorporate the useful features of the object model, it can serve us equally well for many years more. Others would argue that the way ahead lies with the object model, which is in the process of developing a consistent conceptual basis and acquiring the features that the market demands of industry-strength database management systems. Or perhaps some would even argue that database management systems are receding in relative importance amidst the new wealth of middleware capabilities, and the cost of any change at all to the status quo is unlikely to be justified.

So does the database landscape need to change? If so, does the associative model have a role to play in that change? This chapter examines the key tangible business benefits that the associative model offers.

Omnicompetent Programming

The escalating cost of application development has brought about a significant reduction in the number of users developing their own applications. In the days of host-based, green-screen applications, software development was simpler, cheaper and less risky than it is today. It was not unusual for quite modestly-sized companies to develop custom applications whose scope might extend up to that of today’s Enterprise Resource Planning (ERP) systems. Many users regarded packages as a compromise: if applications could enhance competitive edge, it followed that using the same package as your competitors would sacrifice an opportunity to outpace them. But attitudes have changed. In parallel with rising development costs, the range of industry-
specific application packages has expanded, so today custom
development is usually only considered when no suitable
package can be found, and then only by large companies with
correspondingly deep pockets.

Most organisations would be better served by custom
applications than by packages. Successful companies achieve
their pre-eminence by virtue of the ways in which they differ
from their competitors, not the ways in which they resemble
them. Yet dependence on packaged applications has a
homogenising effect on companies that use them.

Market leader SAP’s flagship R/3 ERP package comes
complete with a Reference Model or Business Blueprint, which
claims to embody twenty-five years of best-business practices in
many different industries. SAP encourages its customers to re-
engineer their businesses around the R/3 software, and not to
stray from the business blueprint. The notion that it is
appropriate to change a successful business to fit the dictates of
a software package would have been unacceptable just a few
years ago: now it is accepted as conventional wisdom.

The business blueprint philosophy is flawed. Business
blueprints reflect good historical practice, but more companies
succeed by going beyond conventional wisdom than by
recycling it. The Internet has already taught us that every
established business model is vulnerable to new, wildly
unconventional ways of doing business, facilitated by new
technologies. Moreover, the idea that a software company knows
better than you how your business should operate truly is as silly
as it sounds.

Attitudes are changing. The latter half of the 1990s saw
unprecedented investment in monolithic ERP systems, with
CRM following hard on its heels. Between 1995 and 1999, the
US’s investment in information technology rose by 20% per
year, nearly double the rate in the preceding eight years, but an
investigation by McKinsey whose results were published in
2001 [43] found that this unprecedented rate of investment
generated no productivity improvements in 53 out of 59 economic sectors, leading the London Financial Times to speculate, based on McKinsey’s conclusions, that “Much of the $1,240 billion invested by US business on information technology between 1995 and 1999 may have been wasted.” Also, e-commerce and the advent of web services is creating opportunities for new applications, and those who aim to be at the leading edge are likely to have to build significant pieces of their applications themselves. These factors, coupled with the economic downturn driven by the dot com collapse and September 11th 2001, is prompting CIOs to reassess their investment priorities.

All these factors point to a resurgence of interest in custom applications, which is likely to turn the spotlight back onto the cost and complexity of application development, and thus to reuse. Within the framework of the relational model, software reuse based on the relational model has failed to deliver the benefits that it has promised.

Within the associative model, metacode allows us to write omnicompetent programs 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 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 omnicompetent programs, many end-users will be
able to develop and deploy simple applications without any specialist help.

**Feature Partitioning**

The web is driving new models for application deployment, such as Application Service Provider (ASP) and web services. ASPs and web service providers host applications on their own servers, which are accessed by their customers via the Internet. The challenge faced by service providers and traditional package vendors alike is how to provide the richness of functionality required by sophisticated users without overwhelming those with more prosaic needs.

Historically the behaviour of each installation of an application package has been determined by parameters: the application checks a set of parameter values as it executes to determine precisely how its code should behave in a given installation. This approach has drawbacks, arising from the exponential increase in complexity as new options are added over time. Firstly, 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 it becomes impossible to properly test the full range of configurations created by different combinations of parameters. Also, the number of tables typically proliferates in proportion to the number of options: a full implementation of SAP’s R/3 package contains the equivalent of over 16,500 tables. Finally, 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.
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% of major application package users implement new releases for this reason.

The ASP and web services models add a new dimension to the equation. When each customer has their own version of a 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 service provider must host the package or service for every one of its customers. If a service provider has, say, ten thousand customers using a package or service, does this mean it may find itself hosting up to ten thousand copies 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.

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, childrens’ birthdays or golf handicaps in a company’s enterprise database is unlikely to be deemed acceptable.
The associative model is able to address all of these issues in a natural and unobtrusive way that does not compromise functionality or increase complexity.

As we discussed in Chapter 8, an associative database comprises a number of chapters, and the individual 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.

Here are some scenarios that show how the power of the chapter/profile mechanism may be used to unique advantage:

- 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.

**Instance Schema**

Unlike a relational database, an associative database can readily record data that is unique to one thing of a particular type – one customer, one product and so on – without demanding that it be relevant to all other things of the same type. Also, via the metacode capability, applications that use associative databases can easily present, manipulate and query information that is unique to one particular thing.

In every field of industry, commerce and government, the quality of customer service and how to improve it in a cost-effective manner are near the top of management’s agenda. The media focus on consumer affairs, together with the increasing availability of competitive information and analysis via advertising and the Internet, is encouraging consumers, both individual and corporate, to be more discerning and demanding
in their purchases of both goods and services. In “The One-to-One Future”, Peppers and Rogers claim that a key role of technology will be to put the customer rather than the vendor in charge of the buying process. Applications that can record only a standard menu of data items about each customer leave enterprises ill-equipped to meet such a future.

We have seen how the associative model’s metacode capability allows us to write omnicompetent programs that are able to read and use schemas as they go. When we decide to store a new piece of information about a certain type of entity, we simply add it to the schema. The need to modify programs each time we add a column has gone away: the new information can be immediately understood and processed by our omnicompetent programs.

For example, suppose Avis is a new sales prospect who insists that, if we want its business, we must guarantee that its account balance with us will never exceed $10,000. No customer has ever asked for such a guarantee, so it isn’t something that we currently support or that we want to offer more broadly, but we desperately want Avis’s business. The answer is amend the schema solely for Avis to define the new limit:

\[
\text{Avis has balance limit \textbf{Monetary value}}
\]

\[
\text{Avis has balance limit $10,000}
\]

The Monetary value field will thus appear only on edit panels for Avis, so there is no risk that staff will begin to offer the capability more widely, or that any code we may create to operate on the balance limit will be applied more broadly that it needs to be. In the relational world, an enhancement like this for a single customer would simply be uneconomic, so we would lose the Avis account.
Schema Aggregation

For a programmer to enable users to view two relational databases together as though they were one requires a deep understanding of both databases and a significant amount of difficult programming. Each database has its own, unique data model. Even the most commonplace entities, such as customer, invariably have a different structure and different attributes based on different domains in different databases. It is as though two databases addressing the same problem domain were written in two entirely different languages – simply establishing and sharing what each one means by a concept as fundamental as a customer is almost prohibitively difficult.

And yet the need to view two databases as though they were one is an everyday requirement. Some examples: two applications have been developed separately, and now a third application that spans data used by both is to be developed; two companies are considering a merger, and want to know what proportion of their customers they have in common; the accounts of two subsidiary companies running on different systems need to be consolidated with their parent; a company has been acquired and its systems need to be subsumed into those of its new parent.

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 based on different domains 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?

Let’s take the simplest case of two tables that clearly perform the same function: the two customer tables. We can’t
simply add the rows 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 we have to examine both tables and match up the corresponding columns. Even when we find columns whose functions clearly match, often they will be based on different domains. One designer may have chosen to identify customers by a number, and the other by an alphabetic string. We must choose one or the other, assign a new key to the one that we don’t choose, and then trawl through the entire database replacing foreign keys that point to the keys that we have replaced. 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 that we have just described will often be the tip of the iceberg. Even when we stop short of merging databases – perhaps we need only to answer a simple question such as how many customers the two subsidiaries share in common – we have to go through this cross-referencing exercise before we can begin to find the answer to the question.

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 doesn’t help us to overcome our problems. If we administer the network as one database, we gain no benefit in return for the overhead other than some leeway in resource utilisation. If we administer it as separate databases, we are right back where we started. Distributed database capabilities typically solve only tactical problems, such as allowing one database to access to a table that already exists in another.
In “Data Warehouse – from Architecture to Implementation” [36], Barry Devlin defines a data warehouse as “a single, complete and consistent store of data obtained from a variety of sources and made available to end users in a way they can understand and use in a business context.” The discipline of data warehousing emerged in the 1990s, driven by the realisation that enterprises today often have large and increasing amounts of data but relatively little information. In many organisations, the computer systems infrastructure typically evolves in a piecemeal fashion, often driven by tactical needs within departments. The personal computer has exacerbated this tendency by allowing individuals to create small applications in spreadsheets and personal databases for use by themselves or their workgroup. This bottom-up evolution of infrastructure tends to create disjoint islands of information that are never connected or correlated.

It is issues like these that have spawned the data warehousing industry, which provides tools and techniques to extract data from many individual databases and gather it together into a single, cross-referenced central database that can be used for query and analytical purposes. But there are drawbacks. A data warehouse is costly to set-up and maintain, in terms of specialised human resources, tools that must be purchased and hardware to duplicate all the data. Also the process can be so time-consuming that the data in the warehouse is weeks or months old. Whilst it can still be very useful, as competition increases reaction time becomes ever more important, and managers do not always trust data that is less than current.

Combining two relational databases is like trying to combine two books written in different languages: before you can start on the useful work you have to translate one of them into the language of the other. As we saw above, before we can even begin to combine or correlate two different relational databases we must find and compare matching tables and
columns, resolve differences, and decide what to do about tables and columns that simply don’t match. By contrast, combining two associative databases is like putting together two documents both written in English on the same word processor. You can immediately add one 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 edit and enhance the combined whole to establish common definitions and remove ambiguities.

Associative databases can always be combined with no preparation because the associative model uses one consistent form – items and links – for all types of 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 have used 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.

The predicted shift to web services adds a new urgency to this important aspect of database technology. The associative model is particularly relevant to a web services environment. Web services requires a unified, real-time view of a data universe that selectively aggregates schema and data from a number of individual databases, running in different locations and platforms. Also, the concepts of supertypes, subsets, type equivalence and instance superseding are all needed to create meaning in a data universe constructed from disparate sources. Moreover, the relational primary key mechanism doesn’t support the concept of absolute identity within a federated data universe where different service providers are very likely to have chosen different primary key attributes, whereas the associative model’s concept of absolute identity for both types and instances makes it easy to aggregate instances of similar types from different databases. Associative technology is also designed to manage duplicate types and instances, such as will inevitably occur when a single view of data from different sources is constructed.

**The Time Dimension**

Increasingly industry is acknowledging the cost of collecting data, and its value once it is collected. Unless designers and programmers go to extraordinary lengths, relational databases are capable of presenting only a snapshot of data, not a movie, and must necessarily discard valuable data as they go.

If a company whose details are stored in a relational database changes its address, when their record is updated in the database, the new address replaces the old one, and no record that the address has been changed is visible to the user. In other words, the new data overwrites and destroys the old data.
Before:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ Ltd</td>
<td>125 Canary Wharf, London E2 7YY</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>Customer</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ Ltd</td>
<td>Montague Court, London EC3 4RR</td>
</tr>
</tbody>
</table>

When the same operation is performed in an associative database, the change is effected by marking the association to the old address as changed (by the addition of another association). If their details are held in an associative database, the association to the old address is flagged as no longer current, and a new link to the new address is created. The old link and the old address both remain in the database, and are visible using appropriate software capabilities. No data is overwritten or destroyed.

Before:

**XYZ Ltd** address 125 Canary Wharf, London E6 5TT

After:

**XYZ Ltd** address 125 Canary Wharf, London E6 5TT

... deleted by Transaction 97756392

**XYZ Ltd** address Monatgue Court, London EC3 4RR

... created by Transaction 97756392

Looking under the covers, a relational database comprises two parts: the database itself, which is a snapshot at one instant in time, and the log, or journal, which is a record of all transactions that have changed the state of the database. Specifically, the log contains the “before” and “after” images of changed tuples, and images of new and deleted tuples, together with information about when, how and by whom each change was made. In a sense, one might say that the log is the true record and the
database is merely a snapshot of the record as at some point in time.

All data retrieval, update and query operations are performed solely on the database itself, not on the log (although of course updates are reflected in the log after the event). This representation of data in snapshot form is less than ideal from an operational standpoint. For example, when an enterprise changes its mailing address or telephone number, it will typically notify everyone with whom it deals several weeks ahead of the time at which the change becomes effective. The relational model is not equipped to accept transactions ahead of their effective time and apply them when the effective time is reached. Certainly the relational model can be used to build applications that behave in this way, but such behaviour is not a natural feature of the model. To incorporate it throughout an application would add a substantial overhead of effort and complexity to the development process, and it is not common practice to do so. Typically users devise clerical systems to ensure that transactions are applied at the point in time at which they are effective.

Nor is the relational model equipped to be able to allow users readily to view the state of the database, or of a particular object in it, as at an arbitrary point in time, past, present or future. So it is not evident to users, for example, that a customer has recently changed its address, or what its previous address was. There are many operational circumstances where this type of historical information, were it readily and routinely available, would be useful, and add depth to customer relationship management capabilities. Again, the relational model can be used to build applications that behave in this way, but again it is at the cost of extra effort and complexity, and such refinement is not the norm. As customer service standards and expectations rise, the lack of these two important capabilities is becoming more keenly felt.
Business Rule Automation

Those responsible for the vital day-to-day administration in most modern businesses rely heavily on computer systems to increase their productivity. The functions performed by these systems range from simple record keeping systems that simply store, index and query data, through transaction processing systems, that may capture sales orders, allocate stock and automate despatch, to sophisticated functions such as resource planning, where systems use data from many disparate sources to forecast demand and schedule production.

Yet despite their heavy reliance on computer systems, managers cannot directly influence the way these systems behave. Instead, managers must communicate their requirements to technical specialists, who then translate those requirements into database schemas, application programs and user interfaces. If the auto industry had evolved in the same way, none of us would be capable of driving our own cars and we would all have to employ chauffeurs. Most small and medium-sized enterprises today find themselves wholly reliant on packaged software applications. Extending the motoring analogy, these enterprises cannot afford to hire their own chauffeurs and so are forced to rely on public transport. Anyone who lives in the UK or the U.S. will understand the implications.

Managers haven’t been able to influence the behaviour of the computers that serve them because they haven’t had the right controls: imagine that, instead of pressing an accelerator to speed up your car, you had to reach inside the engine bay and open the carburettor’s butterfly valve. Under such circumstances, you might be forgiven for concluding that it was preferable to have a technician do it for you. The auto industry realised early on that its success lay in producing machines that didn’t require specialist operators, so it worked to develop a set of controls well-suited to end-users. Because the IT industry has been under no such imperative, it has never done so. The
controls that are needed are provided by business rule automation.

Programs that work with relational databases contain an ungainly mixture of procedural logic, which controls the program’s interaction with the user and the database and varies little between programs performing the similar functions over different types of data, and business rules, which deal with the specifics of maintaining the integrity of data and automating business processes. Omnicompotent programming and business rule automation together allow procedural logic and business rules to be separated for the first time. This opens the door to two major steps forward in programming practice: an order of magnitude increase in development productivity, and a new generation of applications that give business managers a significant level of control over the applications that serve them.
APPENDIX 1: THE BOOKSELLER PROBLEM

This appendix illustrates the metadata for the bookseller problem in associative and relational forms side by side. First, the associative model metadata again:

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, here is how SQL would express the same problem for the relational model:

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 metadata takes 51 lines of SQL. Here are the relations that record the same data as the associative model example above:

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Person id</td>
<td>Person name</td>
<td>Country id</td>
</tr>
<tr>
<td>P123</td>
<td>Simon Williams</td>
<td>GB</td>
</tr>
<tr>
<td>P234</td>
<td>Mary David</td>
<td>USA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Country</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Country id</td>
<td>Country name</td>
<td></td>
</tr>
<tr>
<td>GB</td>
<td>Britain</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>America</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Book</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Book id</td>
<td>Book name</td>
<td></td>
</tr>
<tr>
<td>B345</td>
<td>Dr No</td>
<td></td>
</tr>
<tr>
<td>B456</td>
<td>Spycatcher</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Legal entity</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Legal entity id</td>
<td>Legal entity name</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>L02</td>
<td>Bookpages</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Books sold</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Legal entity id</td>
<td>Book id</td>
<td>Points</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>B345</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>B456</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>L02</td>
<td>B345</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>L02</td>
<td>B456</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Books sold by country</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Legal entity id</td>
<td>Book id</td>
<td>Country id</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>B345</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>B345</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>B456</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>L01</td>
<td>B456</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>L02</td>
<td>B345</td>
<td>GB</td>
<td></td>
</tr>
<tr>
<td>L02</td>
<td>B345</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>L02</td>
<td>B456</td>
<td>USA</td>
<td></td>
</tr>
</tbody>
</table>
### Price list

<table>
<thead>
<tr>
<th>Legal entity id</th>
<th>Book id</th>
<th>Country id</th>
<th>Effective date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>L01</td>
<td>B345</td>
<td>GB</td>
<td>1-Jan-98</td>
<td>£10</td>
</tr>
<tr>
<td>L01</td>
<td>B345</td>
<td>USA</td>
<td>1-Mar-98</td>
<td>$16</td>
</tr>
<tr>
<td>L01</td>
<td>B456</td>
<td>GB</td>
<td>1-Jun-98</td>
<td>£7</td>
</tr>
<tr>
<td>L01</td>
<td>B456</td>
<td>USA</td>
<td>1-Jun-98</td>
<td>$12</td>
</tr>
<tr>
<td>L02</td>
<td>B345</td>
<td>GB</td>
<td>1-Jan-98</td>
<td>£8</td>
</tr>
<tr>
<td>L02</td>
<td>B345</td>
<td>USA</td>
<td>1-Jan-98</td>
<td>$14</td>
</tr>
<tr>
<td>L02</td>
<td>B456</td>
<td>USA</td>
<td>1-Apr-98</td>
<td>$13</td>
</tr>
</tbody>
</table>

### Customer

<table>
<thead>
<tr>
<th>Legal entity id</th>
<th>Person id</th>
<th>Points earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>L01</td>
<td>P234</td>
<td>750</td>
</tr>
<tr>
<td>L02</td>
<td>P123</td>
<td>1,200</td>
</tr>
</tbody>
</table>

### Order

<table>
<thead>
<tr>
<th>Order id</th>
<th>Legal entity id</th>
<th>Person id</th>
<th>Book id</th>
<th>Order date</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>O2001</td>
<td>L01</td>
<td>P123</td>
<td>B345</td>
<td>10-Oct-98</td>
<td>£10</td>
</tr>
</tbody>
</table>
APPENDIX 2: REFERENCES AND BIBLIOGRAPHY


INDEX

abstract .......................... 103, 113
abstraction ....................... 5, 135
abstraction flag ............... 118
AFC ...................................... 12
ANSI-SPARC ................. 67
ASP ........................................ 246
aspect ................................. 115
association ...... 84, 87, 116, 222
association type .......... 94, 113
associative algebra ......... 169
associative model ...... 5, 77, 219
atomicity ............................. 128
attributes .............................. 90
Bachman, Charles .... 13, 18, 22
binary model ........ 211, 212
binary relational .......... 211
Boundary Problem .......... 59
business process re-engineering ........................................ 56
cardinality ............. 99, 119, 239
Cartesian product .......... 44
chapter .................. 109, 121, 248
chapter identifier .......... 115
checkpoint ......................... 130
class ........................................ 49
closure ................................. 171
CODASYL .................. 18, 67
Codd, Ted ... 20, 22, 23, 42, 222, 225
column name ..................... 36
commit link ......................... 127
component software .......... 138
concurrency ....................... 130
data independence .......... 14
data model ......................... 17
data sub-language ........ 21
database .............................. 13
database management system 13
datatype .............................. 106
date, Chris .......................... 28, 225
DBTG ................................. 18
dCOM .................................. 61
DDL ..................................... 48
default target ................. 101
default value .................... 240
dependency link .............. 119
difference ......................... 44
dirty read ......................... 131
divide .................................... 44
DML .................................... 48
domain ............................... 36, 238
durability ........................... 129
documentation ..................... 49
entity ..................... 84, 114, 222
delivery statement .......... 94, 113
deployment statement .......... 211
equivalence assertion ....... 120
extend ................................. 46
foreign key ................. 21, 37
fourth generation languages ... 5, 135
...
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>unicompetent</td>
<td>141</td>
</tr>
<tr>
<td>union</td>
<td>44</td>
</tr>
<tr>
<td>Universal Server</td>
<td>2, 27</td>
</tr>
<tr>
<td>Universe of Discourse</td>
<td>14, 73</td>
</tr>
<tr>
<td>unrepeatable read</td>
<td>131</td>
</tr>
<tr>
<td>update</td>
<td>124</td>
</tr>
<tr>
<td>user verb</td>
<td>114</td>
</tr>
<tr>
<td>variables</td>
<td>177</td>
</tr>
<tr>
<td>verb</td>
<td>77, 114</td>
</tr>
<tr>
<td>visibility layer</td>
<td>128</td>
</tr>
<tr>
<td>web services</td>
<td>241, 255</td>
</tr>
<tr>
<td>Wong, Eugene</td>
<td>22</td>
</tr>
<tr>
<td>XML</td>
<td>29, 178</td>
</tr>
<tr>
<td>XML database</td>
<td>30</td>
</tr>
<tr>
<td>XML schema</td>
<td>179</td>
</tr>
</tbody>
</table>