Chapter 2: In Search of a Data Algebra

We all want progress, but if you're on the wrong road, progress means doing an about-turn and walking back to the right road; in that case, the man who turns back soonest is the most progressive.

~ C. S. Lewis

THE ALGEBRA OF DATA was invented by a mathematics professor who was an early employee of Algebraix Data Corporation, a company that was established to develop software products and services based on data algebra. The founders of Algebraix Data were software engineers.

At the time the company was founded, the notion of a data algebra was only a collective gleam in the founders' eyes. The data algebra that was gradually built and mathematically tested in a variety of data processing contexts took about 5 years to evolve. In parallel with this, the company worked on building data management technology that employed the evolving algebra.

The fact that a data algebra never existed until a handful of years ago will likely surprise you. If you are not a software engineer, you probably assume that such an algebra was developed many decades ago and is regularly employed by software engineers to resolve tricky engineering problems they encounter. Alternatively, you may be a software engineer, in which case you may believe that the universe of data, in its extraordinary variety, is simply too complex and convoluted for mathematics to be usefully employed. Both these perspectives are wrong.

The original concept of what a computer could be was mathematical. It was a concept invented by mathematician, Alan Turing, in 1936. Later, he made important contributions to the design of the electromechanical and electronic code-breaking machines that were built and successfully employed during the Second World War. These machines were, in reality, single purpose computers.

However, in the wake of the Second World War, mathematics played only a subsidiary role in the evolution of computers. Most of the advances in computer hardware and computer software that occurred in those years were achievements of electronic engineering. This unfolded in the following way.

In 1945, mathematician and physicist John von Neumann (and others) wrote a paper entitled ***First Draft of a Report on the EDVAC***. EDVAC was an acronym for Electronic Discrete Variable Automatic Computer. During the war, von Neumann had also been working on difficult mathematical problems, particularly the calculation of artillery tables to better enable the effectiveness of artillery. Electronic calculators had been used to great effect for that purpose.

In the paper, von Neumann described what has come to be called "The von Neumann computer architecture." This pragmatic architecture involved a processing unit embodying an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter and memory to store both data and instructions. It also included external mass storage and input and output mechanisms for data. The model, rather than an architecture based on a Turing Machine, established itself at the foundation of the computer industry that gradually emerged.

Mathematicians had some involvement in the further evolution of computers. But there were few efforts to model the processing of data in a mathematical fashion, with data conforming to elements or collections of elements within an algebraic domain and program instructions conforming to algebraic operations on that data – and those efforts were unsuccessful.

Shaving the barber

The barber problem is also known as Russell's paradox, since the famed philosopher and mathematician, Bertrand Russell, popularized it. He used it as a means of pointing out a fundamental problem in Georg Cantor's definition of set theory. The barber problem is this:

> A regiment in the army has a designated barber. The colonel in control of the regiment gives orders to the barber. He commands thus, "From today, every man in the regiment shall be clean-shaven. Henceforth you must shave every man who does not shave himself."

The problem is: who shaves the barber?

Logically, the barber cannot shave himself since he is commanded to shave men who do not shave themselves. And yet if he does not shave himself, he becomes one of the men whom he should shave.

The barber problem is a variation of a familiar Greek paradox (the circular "All Cretans are liars. I am a Cretan"). Its relevance to set theory is that when set theory was originally created by Georg Cantor, a set was deemed to be any definable collection. Russell demonstrated that sets needed to be defined more precisely than that, by considering those sets that are "members of themselves." The problem, which is identical to the barber problem, can be described mathematically as follows:

> Let R be the set of all sets that are not members of themselves. If R is not a member of itself then by definition it must contain itself. But if it contains itself then it violates its own definition as the set of all sets that are not members of themselves. We can express this algebraically as follows:
>
> $$\text{Let } R = \{A\colon A \notin A\}, \text{ then } R \in R \text{ if, and only if, } R \notin R,$$
>
> i.e., let R equal the sets A, such that A is not an element of A, then "R is an element of itself" implies "R is not an element of itself" and vice versa.

Mathematically, the consequence of the barber problem was that the axioms of set theory were reviewed and changed so that such paradoxes were eliminated. Subsequently, set theory became a foundational branch of mathematics, partly because most mathematical objects could be represented within set theory. All data, whatever form it takes, can also be represented as a set or an element of a set.

Functional programming

Despite the computer's evolutionary path, computational mathematics did not die simply because the von Neumann architecture became dominant. In fact, a good deal of work was done on what came to be called lambda calculus (λ-calculus), which in turn played an important role in the theory of programming languages.

We will not describe lambda calculus here, but nevertheless we feel obliged to draw attention to the Church-Turing Thesis, which states that "the untyped lambda calculus is capable of computing all effectively calculable functions." Put simply, it means that mathematicians have a computational model that they believe to be complete.

A natural consequence of the invention of lambda calculus was its use in development of various programming languages. Such languages are generally referred to as functional programming languages, and they resemble algebraic activity by virtue of their applying functions to carry out calculations and transformations. In other words, functional programming languages process data in a mathematical way. What they are missing is a data algebra to define the data.

The tower of Babel

Almost everyone who enters the software world quickly becomes astonished at the number of different programming languages that exist. It has even been suggested that there are now more programming languages than there are human languages. For the record, there are about 6000 human languages, but only about 100 of those languages are spoken by significant populations of people.

The same could be said of programming languages. There are certainly thousands. Wikipedia lists 600 or so languages that it regards as notable. If you read through the list you may encounter many that you have never heard of before, and yet there are many, many more, thousands more, that never made the cut.

Programming languages proliferate for a variety of reasons. Often innovations in computer hardware provoke the creation of new languages. Sometimes the need to specialize a language for a particular area of application (statistics, graphics, video rendering, text processing, etc.) forges a new language. Sometimes software companies wish to create a proprietary language they can control. Sometimes it's simply a matter of evolution. New problems are encountered, new features are added, and thus new dialects are born. Some languages (notably object-oriented ones) are extensible, and hence they evolve of their own accord.

The evolution of functional languages

The first language with functional features was Lisp, developed by John McCarthy (at MIT) in the late 1950s. Other languages such as IPL and APL followed in its wake. In 1977, John Backus formally introduced the idea of functional programming in a Turing Award lecture entitled: *Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs*. He defined functional programs as being constructed in a hierarchical way by means of "combining forms" that allow an "algebra of programs." His paper generated enough interest to provoke research into functional programming.

This gave rise to a whole raft of new programming languages, including: ML, SASL, Miranda, NPL, Hope, J, K and Q. Then, in 1987, there was a general move to form an open standard for functional programming, and the Haskell language was born. Haskell, an open source language, has been evolving ever since then.

Functional programming languages work in a particular way. They are ***declarative***. This means that the program tells the computer what to do without telling it how to do it. This is in contrast to the procedural or imperative style of programming, where the language specifically controls the flow (the procedure) of the computer's activity.

Declarative is an umbrella term in the sense that it covers a number of different language behaviors. In the case of functional programming languages, the language implementation tries to ensure that there are no side effects to the execution of any functions (bugs in procedural programs often turn out to be unexpected side effects of the program code).

So a functional program is a collection of functions that will be executed as specified by the programmer. Ideally, all state changes to any collections of data will be represented as functions. All of this puts an onus on the compiler to resolve how the computer will execute the program. However, it should make it much easier for a programmer to understand and predict the behavior of a program he or she writes.

Hybrid languages

Although the solution of programming problems may in theory be best addressed by thinking of the problem in algebraic terms and formulating a solution by writing algebraic functions, with some applications, it is really

quite difficult for programmers to write code in that manner. This has caused a natural objection to a functional programming language.

For example, algebraic functions are natural features of the R language, which focuses on statistical calculations and algorithms, and the Julia language, which focuses on scientific and high performance computing. It is less obvious why you would need such capability in languages like Perl, Javascript, Visual Basic or even the almost defunct COBOL. All are general purpose languages that are more likely to be used for text processing or transactional applications than algebraic manipulations.

Perhaps you can get the best of both worlds with a language that implements function calls after the fashion of a functional programming language but also has procedural features for handling situations where writing procedural code (telling the computer how to do something) is easier for the programmer.

There are, in fact, many such hybrid languages, including Python, Ruby, Scala, Java and Erlang. Among these, Python is the most popular, which is one of the reasons that Algebraix Data Corporation has chosen to use it as the basis of its Algebraix language.

And Yet, No Data Algebra...

Our brief foray into the evolution of functional programming languages shows that there have been pioneers who tried to exploit mathematical ideas in software. It is well known that the dominance of the von Neumann architecture led to a less mathematically oriented computer industry, but what the practical consequences were and what to do about it has never been clear.

The enthusiasm for functional programming was and is an enthusiasm for mathematics. It was fostered by individuals who were convinced that there were significant advantages to a mathematical style of programming. And there is virtue in it. When such a programming style is enshrined in a programming language, and competently employed, it gives rise to fewer programming errors, requires fewer lines of code and promotes software reuse. In short, it can improve productivity significantly.

Nevertheless, while there have been efforts to mimic mathematics in software, there has been no successful attempt to define an explicit algebra of data that enables data to be dealt with in a mathematical fashion.

The relational malaise

It can be argued that the relational database movement squashed the possibility of data algebra for many years by a process of camouflage. It was born of good intentions. As far as we can tell from the historical record, it was a genuine attempt to introduce a data algebra – the so-called "relational algebra." However, if it ever had a sound mathematical foundation, it soon sacrificed it.

The fundamental idea underlying relational database is to organize data into tables (often called relations) of rows and columns. In general, every entity (a thing: person, product, order, invoice, etc.) has its own table and the various attributes that describe it are columns in the table. Each row in the table represents one instance of the entity (one specific person or product or whatever). In practice, not all tables represent entities, but those that do assign a unique key to each row. This is a physical implementation tactic that ensures the uniqueness of rows, and is useful for linking one entity to another (for instance linking a customer to an order). When used in this way, the unique key is described as a foreign key and is often used to join tables together.

All of this works effectively for data that fits well into tables. If relational database theory had confined itself to managing data in tables, then it is possible that a valid, but very limited, algebra could have been constructed around it. It could, perhaps, have based itself on matrix algebra (a matrix is a table or rectangular array of numbers, symbols, or expressions, arranged in rows and columns).

Unfortunately, that possibility was demolished by the introduction of SQL as a standard query language and, particularly, by the introduction of the null value, a wholly unmathematical idea. This deserves a little explanation.

Practically speaking, in a relational database, a null value can mean any of the following things:

- The value does not exist. One example might be the Social Security Number (SSN) of a U.S. resident who has not yet been issued with such a number – a newborn baby, for example.
- The value was not provided. An example could be the SSN of a U.S. resident who chooses not to provide such information.
- The value is not yet known (i.e., it could be anything). An example is the SSN of a U.S. resident who has not yet provided the database with a value for this attribute.
- The attribute is not applicable to this instance. An example is the SSN of someone who is neither a U.S. citizen nor a U.S. resident.

The first three of these situations could be kluged by assigning specific values to indicate what the context of the missing value is. And it is indeed a kluge since it changes the assigned meaning of the data item SSN. It would now mean "either an indicator denoting the status of this item or an SSN." An alternative kluge would be to assign a zero value to all three, bundling all three of these meanings together under the heading of "we do not know."

The last null possibility is far more troubling. If an entity has attributes that do not apply, then it genuinely does not have those attributes. For example, if we create a table of "living things" and include as an attribute of this table "color of beak," then many rows in the table (almost all mammals) will have null values for this attribute.

Thus, a table is the wrong structure to represent such a collection of data. If you suspect that this is a little synthetic as an example, consider instead a product table for a building materials retailer. Create such a product table, and many products will have attributes that do not apply. Paint may be sold by specific color, an attribute that is not applicable to many other products. Wood may be sold by length, thickness and width. Sand and cement may be sold by weight, and so on. There is undoubtedly an entity called product, but it cannot be represented accurately as a table. When forced under duress into a table, the table becomes populated by a great number of nulls.

If we wanted to represent this mathematically, the nulls would have to disappear, because they represent something that doesn't exist – and with their disappearance, the convenient table structure would also disappear. Instead it would become a collection of products with varying attributes.

The null is schizophrenic, possibly indicating an absent value or possibly indicating that the attribute simply does not apply. This fundamental mathematical error might not have had such a damaging impact if it were not for SQL. SQL had to deal with the nulls. In doing so, it invented a variety of rules which forced relational database further and further away from mathematical legitimacy.

These were not the only failings of the relational approach. Tables are far too limiting to be the only allowable data structures. Thus, other databases emerged (document database, XML database, RDF database, etc.) to cater to other common data structures (hierarchical data, text, graphs, etc.). However, these databases have no mathematical legitimacy either.

The consequences of bad algebra

We could make many criticisms of the relational database from a mathematical standpoint, but we think we have said enough. In the early days of the relational database movement, advocates for the technology were even emboldened to proclaim that the relational approach to data was mathematically correct, when in reality, it wasn't mathematical at all.

When data could be shoe-horned into tables, relational databases worked reasonably well. When the data didn't fit so nicely, there were problems. The idea that there is a right way to structure data (i.e., in tables) is preposterous. In truth, there are simply useful ways to structure data. Sometimes data is best structured as a graph (such as a network of relationships),

sometimes as a recursive structure (such as a bill of materials), sometimes as an ordered list, and so on. Relational databases do some of these things very poorly.

Soon after relational databases became dominant, object-oriented programming languages came to the fore. These languages, which soon became quite popular, did not represent data in a relational fashion at all. This created an awkward problem when programmers chose to store data. They were obliged to translate the data as they had represented it in their program into the form the database required. This problem was called the impedance mismatch. It was so prevalent that an open source mapping capability, Hibernate, was written and widely applied to address the problem.

Thus, an IT industry emerged where programming languages and databases were incompatible in the way they modeled data. Clearly such a situation could not persist indefinitely. Nevertheless, the initial attempt to establish object databases (to complement object-oriented languages) failed commercially, and it wasn't until a second generation of object databases – this time called document databases – emerged that relational databases experienced any competition. However, these relatively new databases have no foundation in a data algebra either.

Axiomatic extended set theory

When E. F. Codd set out to formulate relational theory, he based some of his ideas on extended set theory, which was an idea formulated and described in a 1968 paper, *Description of a Set-Theoretic Data Structure* by D. L. Childs. Childs correctly identified set theory as the natural mathematical basis for representing data and proposed an extended set theory as a means to creating an algebra of data. As a consequence, it would be possible to query data using set operations such as union, intersection, Cartesian product and so on. If this were done, the use of sets and set operations would provide complete independence from physical data structures.

When Algebraix Data Corporation began work on building a data algebra, axiomatic extended set theory, as expounded by D. L. Childs, was examined to see if it was fit for this purpose. It was soon concluded that it constituted a misguided effort.

The assumption on which the genesis of data algebra moved forward was that it should be possible to derive data algebra directly from Zermelo-

Fraenkel set theory (ZFC) without need for any extension. This proved to be correct. The algebra of data described and explained in this book does indeed derive directly from ZFC. Childs' extension was unnecessary.

Nevertheless, teasing out such a universal data algebra from ZFC was not plain sailing. If it had been, an algebra of data would have been propounded decades ago, and it would quickly have become one of the foundations of software.

A final thought

Our intention in this chapter was to provide a brief review of where mathematics has been employed in software engineering at a foundational level. There are, of course, many programs and software products that implement mathematical techniques and formulae to process data (mainly numerical data) in many ways to excellent effect. Functional programming languages were developed to implement mathematical functions in a formal manner. Yet, despite the fact that data is at times even discussed in terms of sets, there was no successful attempt to implement a set theoretical approach to data – until recently.

Taking Stock

The following bullet points summarize this chapter:

- While born of the mathematical work of Alan Turing, the computer industry did not evolve mathematically. Following the computer architecture proposed by John von Neumann, it soon became the domain of electronic engineers and software engineers.
- There were some attempts to introduce and leverage mathematical ideas in software. They were as follows:
  - The introduction of functional programming languages, the most popular of which is Haskell. Functional programming languages implement a programming style that treats computation as the evaluation of mathematical functions while avoiding changing state and mutable data.
  - E. F. Codd's so-called relational algebra.
  - D. L. Childs' extended set theory.
- None of this resulted in the formulation of an algebra of data. Not even close.