SQL Is Declarative, Not Procedural

IN THE PREFACE I told a short story about FORTRAN programmers who could only solve problems using loops and a LISP programmer who could only solve problems recursively. This is not uncommon because we love the tools we know. Let me tell a joke instead of a story: A mathematician, a physicist, and a database programmer were all given a rubber ball and told to find the volume.

The mathematician carefully measured the diameter and either evaluated the volume of sphere formula or used a triple integral if the ball was not perfectly round.

The physicist filled a beaker with water, put the ball in the water, and measured the total displacement. He does not care about the details of the shape of the ball.

The database programmer looked up the model and serial numbers in his rubber ball manufacturer's on-line database. He does not care about the actual ball. But he has information about the tolerances to which it was made, the expected shape and size, and a bunch of other things that apply to the entire rubber ball production process.

The moral of the story is: The mathematician knows how to compute. The physicist knows how to measure. The database guy knows how to look up data. Each person grabs his tools to solve the problem.

Now change the problem to an inventory of thousands of rubber balls. The mathematician and the physicist are stuck with a lot of

manual labor. The database guy does a few downloads and he can produce rubber ball industry standards (assuming that there are such things) and detailed documentation in court with his answers.

1.1 Different Programming Models

Perfecting oneself is as much unlearning as it is learning.
—Edsgar Dijkstra

There are many models of programming. Procedural programming languages use a sequence of procedural steps guided by flow of control statements (`WHILE-DO`, `IF-THEN-ELSE`, and `BEGIN-END`) that change the input data to output data. This was the traditional view of programming, and it is often called the von Neumann Model after John von Neumann, the mathematician who was responsible for it. The same source code runs through the same compiler and generates the same executable module every time. The same program will work exactly the same way every time it is invoked. The keywords in this model are predictable and deterministic. It is also subject to some mathematical analysis *because* it is deterministic.

There are some variations on the theme. Some languages use different flow control statements. FORTRAN and COBOL allocated all the storage for the data at the start of the program. Later, the Algol family of languages did dynamic storage allocation based on the scope of the data within a block-structured language.

Edsgar Dijkstra (see his archives at *www.cs.utexas.edu/users/EWD/*) came up with a language that was nondeterministic. Statements, called guarded commands, have a control that either blocks or allows the statement to be executed, but there is no particular order of execution among the open statements. This model was not implemented in a commercial product, but it demonstrated that something we had thought was necessary for programming (determinism) could be dropped.

Functional programming languages are based on solving problems as a series of nested function calls. The concept of higher-order functions to change one function to another is important in these languages. The derivative and integral transforms are mathematical examples of such higher-order functions. One of the goals of such languages is to avoid a side effect in programs so they can be optimized algebraically. In particular, once you have an expression that is equal to another (in some sense of equality), they can substitute for each other without affecting the result of the computation.

APL is the most successful functional programming language and had a fad period as a teaching language when Ken Iverson wrote his book *A Programming Language* in 1962. IBM produced special keyboards that included the obscure mathematical symbols used in APL for their desktop machines. Most of the functional languages never made it out of academia, but some survive in commercial applications today. Erlang is used for concurrent applications; R is a statistical language; Mathematica is a popular symbolic mathematics product; and Kx Systems uses the K language for large-volume financial analysis. More recently, the ML and Haskell programming languages have become popular among Linux and UNIX programmers.

Here we dropped another concept that had been regarded as fundamental: There is no flow of control in these languages.

Constraint or constraint logic programming languages are a series of constraints on a problem domain. As you add more constraints, the system figures out which answers are possible and which are not. The most popular such language is PROLOG, which also had an academic fad many years ago when Borland Software (*www.borland.com*) made a cheap student version available. The website ON-LINE GUIDE TO CONSTRAINT PROGRAMMING by Roman Barták is a good place to start if you are interested in this topic (*http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html*).

Here we dropped the concept of an algorithm altogether and just provided a problem specification.

Object-oriented (OO) programming is based on the ideas of objects that have both data and behavior in the same module of code. The programming model is a collection of independent cooperating objects instead of a single program invoking functions. An object is capable of receiving messages, processing data, and sending messages to other objects.

The idea is that each object can be maintained and written independently of any particular application and dropped into place where it is needed. Imagine a community of people who do particular jobs. They receive orders from their customers, process them, and return a result.

Many years ago, the INCITS H2 Database Standards Committee (née ANSI X3H2 Database Standards Committee) had a meeting in Rapid City, South Dakota. We had Mount Rushmore and Bjarne Stroustrup as special attractions. Mr. Stroustrup did his slide show with overhead transparencies (yes, this was before PowerPoint was ubiquitous!) about Bell Labs inventing C++ and OO programming, and we got to ask questions.

One of the questions was how we should put OO features into the working model of the next version of the SQL standard, which was known as SQL3 internally. His answer was that Bell Labs, with all their talent, had tried four different approaches to this problem and they came to the conclusion that it should not be done. OO was great for programming but deadly for data.

I have watched people try to force OO models into SQL, and it falls apart in about a year. Every typo becomes a new attribute or class, queries that would have been so easy in a relational model are now multitable monster outer joins, redundancy grows at an exponential rates, constraints are virtually impossible to write so you can kiss data integrity goodbye, and so forth.

With all these programming models, why should we not have different data models?

1.2 Different Data Models

Consider the humble punch card. Punch cards had been used in France to control textile looms since the early 1700s; the method was perfected by Joseph Marie Jacquard in 1801 with his Jacquard loom.

Flash forward to the year 1890, when a man named Herman Hollerith invented a punch card and tabulating machines for that year's United States Census. His census project was so successful that Mr. Hollerith left the government and started the Tabulating Machine Company in 1896. After a series of mergers and name changes, this company became IBM. You might have heard of it.

Up to the 1970s, the "IBM card" and related machinery was everywhere. The most common card was the IBM 5081, and that part number became the common term for it—even across vendors! The punch card was data processing back then.

The physical characteristics of the card determined how we stored and processed data for decades afterwards. The card was the size of an 1887 United States dollar bill (3.25 inches by 7.375 inches). The reason for that size was simple; when Hollerith worked on the Census, he could get drawers to store the decks of cards from the Department of the Treasury across the street.

The cards had a grid of 80 columns of 12 rows, which could accommodate holes. This was for physical reasons again. But once the 80-column convention was established, it stuck. The early video terminals that replaced the key punch machines used screens with 80 columns of text and 24 or 25 rows—that is, two punch cards high and possibly a line for error messages.

Magnetic tapes started replacing punch cards in the 1970s, but they also mimicked the 80-column convention, although there was no longer any need. Many of the early ANSI tape standards for header records are based on this convention. Legacy systems simply replaced card readers with magnetic tape units for obvious reasons, but new applications continued to be built to this standard, too.

The physical nature of the cards meant that data was written and read from left to right in sequential order. Likewise, the deck of cards was written and read from front to back in sequential order.

A magnetic tape file is also written and read in the same way, but with the added bonus that when you drop a tape on the floor, it does not get scrambled like a deck of cards. The downside of a tape over a deck of cards is that it cannot be rearranged manually on purpose either.

Card and tape files are pretty passive creatures and will take whatever an application program throws at them without much objection. Files are also independent of each other, simply because they are connected to one application program at a time and therefore have no idea what other files look like.

Early disk systems also mimicked this model—physically contiguous storage read in a sequential order, with meaning given to the data by the program reading it.

It was a while before disk systems realized that the read/write heads could be moved to any physical position on the disk. This gave us random access storage. We still have a contiguous storage concept within each field and each record, however.

The Relational Model was a big jump, because it divorced the physical and logical models of data. If you read the specifications for many of the early programming languages, they describe physically contiguous data and storage methods. SQL describes only the behavior of the data without any reference to physical storage methods.

1.2.1 Columns Are Not Fields

A field within a record is defined by the application program that reads it. A column in a row in a table is defined independently of any application by the database schema in DDL. The data types in a column are always scalar and NULL-able.

This is a problem for files. If I mount the wrong tape on a tape drive, say a COBOL file, and read it with a FORTRAN program, it can produce meaningless output. The program simply counts the number of bytes from the start of the tape and slices off so many characters into each field from left to right.

The order of the application program variables in the READ or INPUT statements is important, because the values are read into the program variables in that order. In SQL, columns are referenced only by their names. Yes, there are shorthands like the SELECT * clause and "INSERT INTO <table name>" statements that expand into a list of column names in the physical order in which the column names appear within their table declaration, but these are shorthands that resolve to named lists. This is a leftover from the early days of SQL, when we were doing our unlearning and still had a "record-oriented" mindset.

The use of NULLs in SQL is also unique to the language. Fields do not support a missing data marker as part of the field, record, or file itself. Nor do fields have constraints that can be added to them in the record, like the DEFAULT and CHECK() clauses in SQL.

Nor do fields have a data type. Fields have meaning and are defined by the program reading them, not in themselves. Thus, four columns on a punch card containing 1223 might be an integer in one program, a string in a second program, or read as four fields instead of one in a third program.

The choice of data types is not always obvious. The sure symptom of a newbie programmer is that they blindly pick data types without any research. My favorite example is the use of a "VARCHAR (<magical length>)" declaration for almost every column, where <magical length> is an integer value that their particular implement of SQL generates as a default or maximum. In the Microsoft world, look for 255 and 50 to appear.

As an example of the difference in research versus impulse design, consider trying to sort the sections of this book that use a numeric outline for the sections. If you model the outline numbers as character strings, you lose the natural order when you sort them.

For example:

1.1
1.2
1.3
...
1.10

Sorts as:

1.1
1.10

1.2
1.3
...

When this question appeared in a newsgroup, the various solutions included a recursive function, an external function, a proprietary name parsing function, and an extra column for the sort order.

My solution is to pad each section with leading zeros and hope I never have more than 99 headings. Most publishers have an acceptable maximum depth of five levels.

00.00.
01.00.
01.01.
01.01.02.
etc.

You enforce this with `SIMILAR TO` predicate in the DDL rather than trying to do it in the `ORDER BY` clause in the DML.

```
CREATE TABLE Outline
(section_nbr VARCHAR(15) NOT NULL PRIMARY KEY,
   CHECK (section_nbr SIMILAR TO '[:digit:][:digit:]\.+'),
 ..);
```

When you want to display the section numbers without the leading zeros, use a `REPLACE()` or `TRANSLATE` function in the query. We will get to this principle in a later section.

In 25 words or less, columns are active and define themselves; fields are passive and are interpreted by the application program.

1.2.2 Rows Are Not Records

Rows are not records. A record is defined in the application program that reads it, just like the fields. The name of the field in the `READ` statements of the application language tells the program where to put the data. The *physical* order of the field names in the `READ` statement is vital. That means "`READ a, b, c;`" is not the same as "`READ c, a, b;`" because of the sequential order.

A row in a table is defined in the database schema and not by a program at all. The columns are referenced by their names in the schema and not by local program names or physical locations. That means

"SELECT a, b, c FROM..." is the same data as "SELECT c, a, b FROM..." when the data goes into a host program.

All empty files look alike; they are a directory entry in the operating system registry with a name, a length of zero bytes of storage, and a NIL pointer to their starting position. Empty tables still have columns, constraints, security privileges, and other structures, even though they have no rows. All CHECK() constraints are TRUE on an empty table, so you must use a CREATE ASSERTION statement that is external to the tables if you wish to impose business rules on potentially empty tables or among tables.

This is in keeping with the set theoretical model, in which the empty set is a perfectly good set. The difference between SQL's set model and standard mathematical set theory is that set theory has only one empty set, but in SQL each table has a different structure, so they cannot be used in places where nonempty versions of themselves could not be used.

Another characteristic of rows in a table is that they are all alike in structure and they are all the "same kind of thing" in the model. In a file system, records can vary in size, data types, and structure by having flags in the data stream that tell the program reading the data how to interpret it. The most common examples are Pascal's variant record, C's struct syntax, and COBOL's OCCURS clause.

Here is an example in COBOL-85. The syntax is fairly easy to understand, even if you do not read COBOL. The language has a data declaration section in the programs that uses a hierarchical outline numbering system. The fields are strings, described by a template or PICTURE clause. The dash serves the same purpose as the underscore in SQL.

```
01     PRIOR-PERIOD-TABLE.
  05     PERIOD-AMT PICTURE 9(6)
         OCCURS ZERO TO 12 TIMES
         DEPENDING ON PRIOR-PERIODS.
```

The PRIOR-PERIODS field holds the value that controls how many PERIOD-AMT fields we have. ZERO option was added in COBOL-85, but COBOL-74 had to have at least one occurrence.

In Pascal, consider a record for library items that can be either a book or a CD. The declarations look like this:

```
ItemClasses = (Book, CD);
LibraryItems =
RECORD
```

```
  Ref: 0..999999;
  Title: ARRAY [1..30] OF CHAR;
  Author: ARRAY [1..16] OF CHAR;
  Publisher: ARRAY [1..20] OF CHAR;
CASE Class: ItemClasses
  OF Book: (Edition: 1..50; PubYear: 1400..2099);
     CD: (Artist: ARRAY [1..30] OF CHAR;
END;
```

The ItemClasses is a flag that picks which branch of the CASE declaration is to be used. The order of the declaration is important. You might also note that the CASE *declaration* in Pascal was one of the sources for the CASE *expression* in SQL.

Unions in C are another way of doing the same thing we saw done in Pascal. This declaration:

```
union x {int ival; char j[4];} mystuff;
```

defines mystuff to be either an integer (which are 4 bytes on most modern C compilers, but this code is nonportable) or an array of 4 bytes, depending on whether you say `mystuff.ival` or `mystuff.j [0]`.

As an aside, I tried to stick with the idioms of the languages—all uppercase for COBOL, capitalized name in Pascal, and lowercase in C. COBOL is all uppercase because it was first used on punch cards, which only have uppercase. C was first written on Teletype terminals for mini computers, which have a shift key, but the touch is so hard and so long that you have to hit the keys vertically; you cannot type with your fingertips. C was designed for two-finger typists, pushing the keys with strokes from their elbows rather than the wrist or fingertips. SQL and modern language idioms are based on the ease of text formatters and electronic keyboards that respond to fingertip touch.

Once more, the old technology is reflected in the next technology, until eventually the new technology finds its voice. These styles of formatting code are not the best practices for human readability, but they were the easiest way of doing the job at the time. You can get some details about human factors and readability in my other book, *SQL Programming Style* (ISBN 0-12-088797-5).

The OCCURS keyword in Cobol, `union` in C, and the variant records in Pascal have a number or flag that tells the program how to read a record structure you input as bytes from left to right.

In SQL the *entire row* is read and handled as the "unit of work," and it is *not* read sequentially. You UPDATE, INSERT, and DELETE *entire* rows and not columns within a row. The ANSI model of an UPDATE is that it acts as if

1. You go to the base table (updatable VIEWs are first resolved to their underlying base table). It cannot have an alias because an alias would create a working table that would be updated and then disappear after the statement is finished, thus doing nothing.
2. You go to the WHERE clause. All rows (if any!) that test TRUE are marked as a subset. If there is no WHERE clause or the search condition is always TRUE, then the entire table is marked as the subset. If the search condition is always FALSE or UNKNOWN, then the subset is empty. But an emptyset is still a set and gets treated as such. The name of this set/pseudo-table is OLD in Standard SQL, and it can be used in TRIGGERs.
3. You go to the SET clause and construct a set/pseudo-table called NEW. The rows in the NEW table are built two ways: if they are not on the left side of the SET clause, then the values from the original row are copied; if the columns are on the left side of the SET clause, then the expression on the right side determined their value. This is supposed to happen in parallel for all the columns, all at once. That is, the unit of work is a row, not one column at a time.
4. The OLD subset is deleted and the NEW set is inserted. This is why

```
UPDATE Foobar
   SET a = b, b = a;
```

swaps the values in the columns "a" and "b," while a sequence of assignment statements in a procedural file–oriented language would behave like this:

```
BEGIN
SET a = b;
SET b = a;
END;
```

and leave the original value of "b" in both columns.

5. The engine checks constraints and does a ROLLBACK if there are violations.

In full SQL-92, you can use row constructors to say things like:

```
UPDATE Foobar
  SET (a, b)
    = (SELECT x, y
         FROM Floob AS F1
        WHERE F1.keycol= Foobar.keycol);
```

Think about what a confused mess this statement is in the SQL model:

```
SELECT f(c2) AS c1, f(c1) AS c2 FROM Foobar;
```

The entire row comes into existence all at once as a single unit. That means that "c1" does not exist before the second function call. Such nonsense is illegal syntax.

1.2.3 Tables Are Not Files

There is no sequential access or ordering in table, so "first," "next," and "last" rows are totally meaningless. If you want an ordering, then you need to have a column that defines that ordering. You must use an ORDER BY clause in a cursor or in an OVER() clause.

An RDBMS seeks to maintain the correctness of all its data. The methods used are triggers, constraints, and declarative referential integrity.

Declarative referential integrity (DRI) says, in effect, that data in one table has a particular relationship with data in a second (possibly the same) table. It is also possible to have the database change itself via referential actions associated with the DRI.

For example, a business rule might be that we do not sell products that are not in inventory. This rule would be enforced by a REFERENCES clause on the Orders table that references the Inventory table and a referential action of ON DELETE CASCADE, SET DEFAULT, or whatever.

Triggers are a more general way of doing much the same thing as DRI. A trigger is a block of procedural code that is executed before, after, or instead of an INSERT INTO or UPDATE FROM statement. You can do anything with a trigger that you can do with DRI and more.

However, there are problems with triggers. While there is a standard syntax for them in the SQL-92 standard, most vendors have not implemented it. What they have is very proprietary syntax instead. Second, a trigger cannot pass information to the optimizer like DRI. In the example in this section, I know that for every product number in the Orders table, I have that same product number in the Inventory table. The optimizer can use that information in setting up EXISTS() predicates and JOINs in the queries. There is no reasonable way to parse procedural trigger code to determine this relationship.

The CREATE ASSERTION statement in SQL-92 will allow the database to enforce conditions on the entire database as a whole. An ASSERTION is not like a CHECK() clause, but the difference is subtle. A CHECK() clause is executed when there are rows in the table to which it is attached. If the table is empty, then all CHECK() clauses are effectively TRUE. Thus, if we wanted to be sure that the Inventory table is never empty, we might naively write:

```
CREATE TABLE Inventory
( ...
 CONSTRAINT inventory_not_empty
       CHECK ((SELECT COUNT(*) FROM Inventory) > 0), ... );
```

and it would not work. However, we could write:

```
CREATE ASSERTION Inventory_not_empty
       CHECK ((SELECT COUNT(*) FROM Inventory) > 0);
```

and we would get the desired results. The assertion is checked at the schema level and not at the table level.

A file is closely related to its physical storage media. A table may or may not be a physical file at all. DB2 from IBM uses one physical file per table, while Sybase puts several entire databases inside one physical file. A table is a set of rows of the same kind of thing. A set has no ordering and it makes no sense to ask for the first or last row.

A deck of punch cards is sequential, and so are magnetic tape files. Therefore, a physical file of ordered sequential records also became the mental model for data processing and it is still hard to shake. Anytime you look at data, it is in some physical ordering.

The various access methods for disk storage system came later, but even these access methods could not shake the contiguous, sequential mental model.

Another conceptual difference is that a file is usually data that deals with a whole business process. A file has to have enough data in itself to support applications for that business process. Files tend to be "mixed" data that can be described by the name of the business process to which they belong, such as "the Payroll file" or something like that.

Tables can be entities, relationships, or auxiliaries within a business process. This means the data that was held in one file is often put into several tables. Tables tend to be "pure" data that can be described by single words. The payroll would now have separate tables for time cards, employess, projects, and so forth.

1.2.4 Relational Keys Are Not Record Locators

One of the first things that a newbie does is use a proprietary autonumbering feature in their SQL product as a `PRIMARY KEY`. This is completely wrong, and it violates the definition of a relational key.

An attribute has to belong to an entity in the real world being modeled by the RDBMS. Autonumbering does not exist in an entity in the real world being modeled by the RDBMS. Thus, it is not an attribute and cannot be in a table, by definition.

Autonumbering is a result of the physical state of particular piece of hardware at a particular time as read by the current release of a particular database product. It is not a data type. You cannot have more than one column of this "type" in a table. It is not `NULL`-able, which all data types have to be in SQL. It is not a numeric; you cannot do math with it. It is what is called a "tag number"—basically, a nominal scale written with numbers instead of letters. Only equality tests make sense.

1.2.4.1 Redundant Duplicates

Assume we have a table of vehicles with some autonumbering feature as its key—I will use a function call notation here. Execute this code with the same `VALUES()` clause.

```
INSERT INTO Vehicles (auto_nbr(), vin, mileage, ..)
VALUES ( ..);
INSERT INTO Vehicles (auto_nbr(), vin, mileage, ..)
VALUES ( ..);
```

I now have two cars with the same `VIN` number. Actually, I have two copies of the same car (object) with an autonumber pseudo-key instead

of the industry standard VIN as the proper relational key. This is called an insertion anomaly.

Assume that this pair of insertions led to creating vehicles with pseudo-keys 41 and 42 in the table, which are the same actual object. I can update 42's mileage without touching 41. I now have two versions of the truth in my table. This is a called an update anomaly.

Likewise, if I wreck vehicle 41, I still have copy 42 in the motor pool in spite of the fact that the actual object no longer exists. This is deletion anomaly.

1.2.4.2 Uniqueness Is Ruined

Before you say that you can make a key from (auto-numbering, vin), read more from Dr. E. F. Codd: "If the primary key is composite and if one of the columns is dropped from the primary key, the first property [uniqueness] is no longer guaranteed."

Assume that I have correct VINs and use (auto-numbering, vin) as a key. Dropping the pair clearly does not work—a lot of vehicles could have the same mileage and tire sizes, so I do not have unique rows guaranteed. Dropping the autonumber will leave me with a proper key that can be validated, verified, and repeated.

Dropping the VIN does not leave me with a guarantee (i.e., repeatability and predictability). If I run this code:

```
BEGIN ATOMIC
DELETE FROM Vehicles
 WHERE id = 41;
INSERT INTO Vehicles (mileage, ..)
 VALUES (<<values of #41>> );
END;
```

the relational algebra says that I should have in effect done nothing. I have dropped and reinserted the same object—an EXCEPT and UNION operation that cancel. But since autonumbering is physical and not logical, this does not work.

If I insert the same vehicle (object) into another table, the system will not guarantee me that I get the same autonumbering as the relational key in the other table. The VIN would be guaranteed constant in this schema and any other schema that needs to model a vehicle.

The guarantee requirement gets worse. SQL is a set-oriented language and allows me to write things like this:

```
INSERT INTO Vehicles (pseudo_key, vin, mileage, ..)
SELECT auto_nbr(), vin, mileage, ..
 FROM NewPurchases;
```

Since a query result is a table, and a table is a set that has no ordering, what should the autonumbers be? The entire, whole, completed set is presented to Vehicles all at once, not a row at a time. There are (n!) ways to number (n) rows. Which one did you pick? Why? The answer in such SQL products has been to use whatever the physical order of the physical table happened to be. That nonrelational phrase "physical order" again!

But it is actually worse than that. If the same query is executed again, but with new statistics or after an index has been dropped or added, the new execution plan could bring the result set back in a different physical order.

Can you explain from a logical model why the same rows in the second query get different pseudo-keys? In the relational model, they should be treated the same if all the values of all the attributes are identical and each row models the same object as it did before.

1.2.5 Kinds of Keys

Now for a little more practice than theory. Here is my classification of types of keys. It is based on common usage.

1. A natural key is a subset of attributes that occur in a table and act as a unique identifier. They are seen by the user. You can go to the external reality or a trusted source and verify them. You would also like to have some validation rule. Example: UPC codes on consumer goods (read the package barcode), which can be validated with a check digit, a manufacturer's website, or a tool (geographical coordinates validate with a GPS tool).
2. An artificial key is an extra attribute added to the table that is seen by the user. It does not exist in the external reality, but can be verified for syntax or check digits inside itself. It is up to the DBA to maintain a trusted source for them inside the enterprise. Example: the open codes in the UPC scheme to which a user can assign products made inside the store. The most common example is grocery stores that have bakeries or delicatessens

inside the stores. The check digits still work, but you have to define and verify them inside your own enterprise.

If you have to construct a key yourself, it takes time to design them, to invent a validation rule, set up audit trails, and so forth. Yes, doing things right takes time and work. Not like just popping an autonumbering on every table in the schema, is it?

3. An "exposed physical locator" is not based on attributes in the data model but in the physical storage and is exposed to the user. There is no reasonable way to predict it or verify it, since it usually comes from the physical state of the hardware at the time of data insertion. The system obtains a value through some physical process in the hardware totally unrelated to the logical data model.

   Just because autonumbering does not hold a track/sector address (like Oracle's ROWID) does not make it a logical key. A hash points to a table with the address. An index (the mechanism in autonumbering) resolves to the target address via pointer chains. If you rehash or reindex, the physical locator has to resolve to the new physical location.

4. Surrogate keys were defined in a quote from Dr. E. F. Codd: "...Database users may cause the system to generate or delete a surrogate, but they have no control over its value, nor is its value ever displayed to them ..." (Dr. E. F. Codd in *ACM Transactions on Database Systems*, pp. 409–410), and in Codd, E. F., "Extending the Database Relational Model to Capture More Meaning," *ACM Transactions on Database Systems*, 4(4), 1979, pp. 397–434.

This means that a surrogate ought to act like an index: created by the user, managed by the system, and NEVER seen by a user. That means never used in queries, DRI, or anything else that a user does.

Codd also wrote the following:

> There are three difficulties in employing user-controlled keys as permanent surrogates for entities.
>
> 1. The actual values of user-controlled keys are determined by users and must therefore be subject to change by them (e.g., if two companies merge, the two employee databases might be combined with the result that some or all of the serial numbers might be changed.)
>
> 2. Two relations may have user-controlled keys defined on distinct domains (e.g., one uses Social Security, while

the other uses employee serial numbers) and yet the entities denoted are the same.

3. It may be necessary to carry information about an entity either before it has been assigned a user-controlled key value or after it has ceased to have one (e.g., an applicant for a job and a retiree).

These difficulties have the important consequence that an equijoin on common key values may not yield the same result as a join on common entities. A solution—proposed in Chapter 4 and more fully in Chapter 14—is to introduce entity domains that contain system-assigned surrogates.

> *Database users may cause the system to generate or delete a surrogate, but they have no control over its value, nor is its value ever displayed to them....*
>
> *—Codd, in ACM TODS, pp. 409–410).*

1.2.6 Desirable Properties of Relational Keys

In an article at *www.TDAN.com* by Mr. James P. O'Brien (Maximum Business Solutions), the author outlined desirable properties of relational keys. I agree with almost everything he had to say, but I have to take issue on some points.

I agree that natural keys can be inherent characteristics, such as DNA signatures, fingerprints, and (longitude, latitude). I also agree that the ISO-3779 Vehicle Identification Number (`VIN`) can be a natural key. What makes all of these natural keys is a property that Mr. O'Brien does not mention: they can be verified and validated in the real world.

When I worked for a state prison system, we moved inmates by fingerprinting them because we had to be absolutely sure that we did not let someone out before their time, or keep them in prison longer than their sentence. If I want to verify (longitude, latitude) as an attribute, I can walk to the location, pull out a GPS tool, and push a button. The same principle holds for colors, weights, and other physical measurements that can be done with instruments.

The `VIN` is a bit different. I can look at the format and determine if it is a valid `VIN`—Honda does not make a Diablo and Lamborghini does not make a Civic. However, if the parts of the `VIN` are in the correct format, I need to contact the automobile manufacturer and ask if the `VIN` was actually issued. If Honda made 1,000,000 Civics, then a `VIN` for the 1,000,001th Civic is a fake.

Validate internally, and verify externally. But this leads to the concept of a "trusted source" that can give us verification. And that leads to the question, "How trusted?" is my source.

My local grocery story believes that the check I cash is good and that the address on the check and driver's license number are correct. If I produced a license with a picture of Britney Spears that did not match the name on the check, they would question it. But as long as the photo ID looks good and has a bald white male who looks like "Ming the Merciless" from the old Flash Gordon comic strips on it, they will probably cash the check.

When I travel to certain countries, I need a birth certificate and a passport. This is a higher degree of trust. For some security things I need to provide fingerprints. For some medical things, I need to provide DNA—that is probably the highest degree of trust, since in theory you could make a clone from my sample, *á la* many science fiction stories.

The points I want to challenge in Mr. O'Brien's article are that a natural key

1. Must have an invariant value
2. Must have an invariant format

1.2.7 Unique But Not Invariant

In 2007, the retail industry in the United States switched from the 10-digit UPC barcode on products to the 13-digit EAN system, and the International Standard Book Number (ISBN) is falling under the same scheme. Clearly, this violates Mr. O'Brien's condition. But the retail industry is still alive and well in the United States. Why?

The most important property of a key is that it must ensure uniqueness. But that uniqueness does not have to be eternal. Nor does the format have to be fixed for all time. It simply has to be verifiable at the time I ask my question.

The retail industry has assured that the old and the new barcodes will identify the same products by a carefully planned migration path. This is what allowed us to change the values and the formats of one of the most common identifiers on earth. The migration path started with changing the length of the old UPC code columns from 10 to 13 and padding them with leftmost zeros.

In a well-designed RDBMS product, referenced keys are easy to change. Thus, I might have an Inventory table that is referenced in the

Orders table. The physical implementation is a pointer in the Orders table back to the single value in the Inventory table. The main problem is getting the data types correctly altered.

Mr. O'Brien argues for exposed physical locators when

- No suitable natural key for the entity exists.
- A concatenated key is so lengthy that performance is adversely affected.

The first condition—no suitable natural key exists—is a violation of Aristotle's law of identity (to be is to be something in particular) and the result of a bad RDBMS design flaw. Or the designer is too lazy to look for industry standards.

But if you honestly cannot find an industry standard and have to create an identifier, then you need to take the time to design one, with validation and verification rules, instead of returning to 1950s-style magnetic tape files' use of an exposed physical locator.

The argument that a concatenated key that is "too long" forgets that you have to ensure the uniqueness of that key to maintain data integrity anyway. Your performance choices are to either have the SQL engine produce a true surrogate or to design an encoding that is shorter for performance. The `VIN` has a lot of data (country, company, make, model, plant, etc.) encoded in its 17-character string for verification.

1.3 Tables as Entities

An entity is a physical or conceptual "thing" that has meaning in itself. A person, a sale, or a product would be an example. In a relational database, an entity is defined by its attributes, which are shown as values in columns in rows in a table.

To remind users that tables are sets of entities, I like to use collective or plural nouns that describe the function of the entities within the system for the names of tables. Thus "Employee" is a bad name because it is singular; "Employees" is a better name because it is plural; "Personnel" is best because it is collective noun and does not summon up a mental picture of individual persons, but of an abstraction (see *SQL Programming Style*, ISBN: 0-12088-797-5, for more details).

If you have tables with exactly the same structure, then they are sets of the same kind of elements. But you should have only one set for each kind of data element! Files, on the other hand, were physically separate units of storage that could be alike—each tape or disk file represents

a step in the procedure, such as moving from raw data to edited data, sorting and splitting the data for different reports, and finally sending it to archival storage.

In SQL, this physical movement should be replaced by a logical status code in a single table. Even better, perhaps the RDBMS will change status code for you without your actions. For example, an account over 120 days past due is changed to "collections status" and we send the account holder a computer-generated letter.

1.4 Tables as Relationships

A relationship is shown in a table by columns that reference one or more entity tables. Without the entities, the relationship has no meaning, but the relationship can have attributes of its own. For example, a show business contract might have an agent, a studio, and a movie star. The method of payment is an attribute of the contract itself, and not of any of the three parties.

These tables will always have `FOREIGN KEY` references to the entities in the relationship and DRI actions to enforce the business rules.

1.5 Statements Are Not Procedures

Declarative programming is not like procedural programming. We seek to keep the data correct by using constraints that exclude the bad data at the start. We also want to use data rather than computations to solve problems, because SQL is a data retrieval language and not a computational one.

As an example of the difference, the PASS-2006 SQL Server group conference has a talk on Common Language Resources (CLR) in that product. This is a proprietary Microsoft "feature" that lets you embed any of several procedural or OO languages inside the database. The example the speaker used was putting a Regular Expression object to parse an e-mail address as a constraint.

The overhead was high, execution time was slow, and the regular expression parser called might or might not match the `SIMILAR TO` predicate in ANSI/ISO Standard SQL, depending on the CLR language used. But the real point was that needless complexity could have been avoided. Using a `TRANSLATION` (or nested `REPLACE()` functions if your SQL does not support ANSI/ISO Standard SQL) in a `CHECK()` constraint could have prevented bad e-mail addresses in the first place.

Declarative programming prevents bad data, while procedural programming corrects it.

1.6 Molecular, Atomic, and Subatomic Data Elements

If you grew up as a kid in the 1950s, you will remember those wonderful science fiction movies that always had the word "atomic" in the title, like *Atomic Werewolf from Mars* or worse. We were still in awe of the atomic bomb and were assured that we would soon be driving atomic cars and airplanes. It was sort of like the adjectives "extreme" or "agile" are today. Nobody knows quite what it means, but it sounds really, really cool.

Technically, "atom" is the Greek word meaning "without parts" or "indivisible." The original idea was that if you kept dividing a physical entity into smaller and smaller pieces, you would eventually hit some lower bound. If you went beyond that lower bound, you would destroy that entity.

When we describe First Normal Form (1NF) we say that a data element should hold atomic or scalar values. What we mean is that if I try to pull out "subatomic parts" from the value in a column, it loses meaning.

Scalar is used as a synonym for atomic, but it actually is a little trickier. It requires that there be a scale of measurement from which the value is drawn and from which it takes meaning. It is a bit stricter, and a good database designer will try to establish the scales of measurement in his or her data model.

Most newbies assume that if they have a column in an SQL table, this automatically makes the value atomic. A column cannot hold a data structure, like an array, linked list, or another table, and it has to be of a simple data type. Ergo, it must be an atomic value. This was very easy up to Standard SQL-92, since the language had no support for those structures. This is no longer true in SQL-99, which introduces several very nonrelational "features," and to which several vendors added their own support for arrays, nested tables, and variant data types.

Failure to understand atomic versus scalar data leads to design flaws that split the data so as to hide or destroy facts, much like splitting atomic structures destroys or changes them.

1.6.1 Table Splitting

The worst way to design a schema is probably to split an attribute along tables. If I were to design a schema with a "Male_Personnel" and a "Female_Personnel" table or one table per department, you would see the fallacy instantly. Here an attribute, gender, or department, is turned into metadata for defining tables.

In the old punch cards and tape file system days, we physically moved data to such selective files to make processing easier. It was how we got parallelism and did a selection. The most common split is based on time—one table per day, week, month, or year. The old IBM magnetic tape library systems used a label based on a "yyddd" format—Julianized day within a two-digit year. That label was used to control when a tape was refreshed—magnetic tape degrades over time due to cosmic rays and heat, so tapes had to be reread and rewritten periodically. Reports were also based on time periods, so the physical tapes served the same filtering function as a `WHERE` clause with a date range.

The next most common split is geographical. Each physical location in the enterprise is modeled as its own table, even though they are the same kind of entity. Again, this can be traced back to the old days, when each branch office prepared its own reports on paper, then on punch cards, and then on magnetic tapes for the central office.

A partitioned table is not the same thing. It is one logical, semantic unit of data; the system and not the applications maintain it. The fact that it is physically split across physical file structures has nothing to do with the semantics.

Perhaps the fact that DDL often has a mix of logical data descriptions mixed with physical implementations in vendor extensions confuses us. As an aside, I often wonder if SQL should have had a separate syntax for referential integrity, relational cardinality, membership, domain constraints, and so forth, rather than allowing them in the DDL.

1.6.2 Column Splitting

The other mistake is having an atomic attribute and splitting it into columns. As we all know from those 1950s science fiction movies, nothing good comes from splitting atoms—it could turn your brother into an atomic werewolf!

A phone number in the United States is displayed as three sections (area code, exchange, and number). Each part is useless by itself. In fact, you should include the international prefixes to make it more exact, but usually context is enough. You would not split this data element over three columns, because you search and use this value in the order that it is presented, and you use it as a whole unit. This is an atom and not a molecule.

You can also split a single data element across rows. Consider this absurd table:

```
CREATE TABLE Personnel
(worker_name CHAR(20) NOT NULL,
attribute_name CHAR(15) NOT NULL
   CHECK (attribute_name IN ('weight', 'height',
   'bowling score')),
attribute_value INTEGER NOT NULL,
PRIMARY KEY (worker_name, attribute_name));
```

The bad news is that you will see this kind of thing in the real world. One column gives metadata and the other gives a value.

Look at a subtler version of the same thing. Consider this table that mimics a clipboard upon which we record the start and finish of a task by an employee.

```
CREATE TABLE TaskList
(worker_name CHAR(20) NOT NULL,
task_nbr INTEGER NOT NULL,
task_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
task_status CHAR(1) DEFAULT 'S' NOT NULL
   CHECK (task_status IN ('S', 'F')),
PRIMARY KEY (worker_name, task_nbr, task_status));
```

In order to know if a task is finished (`task_status = 'F'`), we first need to know that it was started (`task_status = 'S'`). That means a self-join in a constraint. A good heuristic is that a self-joined constraint means that the schema is bad, because something is split and has to be reassembled in the constraint.

Let's rewrite the DDL with the idea that a task is a data element.

```
CREATE TABLE TaskList
(worker_name CHAR(20) NOT NULL,
task_nbr INTEGER NOT NULL,
task_start_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
task_end_time TIMESTAMP, -- null means in process
 PRIMARY KEY (worker_name, task_nbr));
```

Temporal split is the most common example, but there are other ways to split a data element over rows in the same table.

1.6.3 Temporal Splitting

The most common newbie error is splitting a temporal data element into (year, month, day) columns or as (year, month) or just (year) columns. There is a problem with temporal data. By its nature, it is not atomic; it is a continuum. A continuum has no atomic parts; it can infinitely subdivide. Thus, the year '2005' is shorthand for the pair ('2005-01-01 00:00:00', '2005-12-31 23:59:59.999 …') where we live with the precision that our SQL product has for the open end on the left. It includes every point in between. That means every uncountable infinite one of them.

The Greeks did not have a concept of a continuum, and this lead to Zeno's famous paradoxes. Hey, this is a database book, but you can Google Greek philosophy for yourself. In particular, look for "Resolving Zeno's Paradoxes" by W. I. McLaughin (*Scientific American*, November 1994).

1.6.4 Faking Non-1NF Data

So, how do programmers "fake it" within the syntax of SQL when they want non-1NF data semantics to mimic a familiar record layout? One way is to use a group of columns where all the members of the group have the same semantic value; that is, they represent the same data element. Consider the table of an employee and his children:

```
CREATE TABLE Employees
(emp_nbr INTEGER NOT NULL,
emp_name CHAR(30) NOT NULL,
...
child1 CHAR(30), birthday1 DATE, sex1 CHAR(1),
child2 CHAR(30), birthday2 DATE, sex2 CHAR(2),
child3 CHAR(30), birthday3 DATE, sex3 CHAR(1),
child4 CHAR(30), birthday4 DATE, sex4 CHAR(1));
```

This looks like the layouts of many existing file system records in COBOL and other 3GL languages. The birthday and sex information for each child is part of a repeated group and therefore violates 1NF. This is faking a four-element array in SQL; the index just happens to be part of the column name!

Very clearly, the dependents should have been in their own table. There would be no upper limit on family size, aggregation would be much easier, the schema would have fewer NULLs, and so forth.

Suppose I have a table with the quantity of a product sold in each month of a particular year, and I originally built the table to look like this:

```
CREATE TABLE Abnormal
(product CHAR(10) NOT NULL PRIMARY KEY,
month_01 INTEGER, -- null means
month_02 INTEGER,
...
month_12 INTEGER);
```

and I wanted to flatten it out into a more normalized form, like this:

```
CREATE TABLE Normal
(product CHAR(10) NOT NULL,
month_nbr INTEGER NOT NULL,
qty INTEGER NOT NULL,
PRIMARY KEY (product, month_nbr));
```

I can use the statement

```
INSERT INTO Normal (product, month_nbr, qty)
SELECT product, 1, month_01
  FROM Abnormal
 WHERE month_01 IS NOT NULL
UNION ALL
SELECT product, 2, month_02
  FROM Abnormal
WHERE month_02 IS NOT NULL
...
UNION ALL
SELECT product, 12, month_12
  FROM Abnormal
 WHERE bin_12 IS NOT NULL;
```

While a `UNION ALL` expression is usually slow, this has to be run only once to load the normalized table, and then the original table can be dropped.

1.6.5 Molecular Data Elements

A molecule is a unit of matter made up of atoms in a particular arrangement. So let me define a unit of data made up of scalar or atomic values in a particular arrangement. The principle characteristic

is that the whole loses precise meaning when any part is removed. Note that I said precise meaning—it can still have some meaning, but it now refers to a set, possibly infinite, of values instead of single value or data element.

One example would be (longitude, latitude) pairs kept in separate columns. Together they give you a precise location, a geographical point (within a certain error), but apart they describe a line or a circle with an infinite number of points.

Yes, you could model a location as a single column with the pair forced inside it, but the arithmetic would be a screaming pain. You would have to write a special parser to read that column, effectively making it a user-defined data type. Making it a "two-atom molecule" makes much more sense. But the point is that semantically it is one data element, namely a geographical location.

Likewise, the most common newbie error is to put a person's name into one column, rather than having `last_name`, `first_name`, and `middle_name` columns. The error is easy to understand; a name is a (relatively) unique identifier for a person and identifiers are semantically atomic. But in practice, sorting, searching, and matching are best done with the atoms exposed.

1.6.6 Isomer Data Elements

The worst situation is isomer data elements. An isomer is a molecule that has the same atoms as another molecule but arranged a little differently. The most common examples are right- and left-handed versions of the same sugar. One creature can eat the right-handed sugar but not the left-handed isomer.

The simple example is a table with a mix of scales, say temperatures in both Celsius and Fahrenheit. This requires two columns, one for the number and one for the scale. I can then write `VIEW`s to display the numbers on either scale, depending on the user. Here the same semantic value is modeled dynamically by a `VIEW`. The correct design would have picked one and only one scale, but bear with me; things get worse.

Consider mixed currencies. On a given date, I get a deposit in one of many currencies, which I need to convert to other currencies, all based on the daily exchange rate.

```
CREATE TABLE Deposits
(..
deposit_amt DECIMAL (20,2) NOT NULL,
```

```
currency_code CHAR(3) NOT NULL, -- use ISO code
deposit_date DATE DEFAULT CURRENT_DATE NOT NULL,
..);
CREATE TABLE ExchangeRates
(..
currency_code CHAR(3) NOT NULL, -- use ISO code
exchange_date DATE DEFAULT CURRENT_DATE NOT NULL,
exchange_rate DECIMAL (8,4)NOT NULL,
..);
```

Semantically, the deposit had one and only one value at that time. But I express it in U.S. dollars, and my friend thinks in euros. There is no single hard formula for converting the currencies, so you have to use a join.

```
CREATE VIEW DepositsDollars (.., dollar_amt, )
AS
SELECT .., (D1.deposit_amt * E1.exchange_rate),
 FROM Deposits AS D1, ExchangeRates AS E1
 WHERE D1.deposit_date = E1.exchange_date;
```

and likewise there will be a "DepositsEuros" with a euro-amt column, and whatever else we need. The VIEWs are good, atomic scalar designs, but the underlying base tables are not!

Another approach would have been to find one unit of currency and only use it, doing the conversion on the front end. The bad news is that such an approach would have lost information about the relative positions among the currencies and been subject to rounding errors. This is not an easy problem.

1.6.7 Validating a Molecule

The major advantage of keeping each atomic data element in a column is that you can easily set up rules among them to validate the whole. For example, an address is a molecular unit of data. Within it, I can see if the city and state codes match the ZIP code.

Instead of putting such constraints into one CHECK() constraint, break it into separate constraints that have meaningful names that will show up in errors messages.

This leads to the next section and a solution for the storage versus processing problem.