CS319: SQL and The Relational Model

The Askew Wall

Notes keyed to slides

1. The Askew Wall

A sometimes light-hearted but always deadly serious investigation, littered with anecdotes, into what the author regards as one of the very worst computer languages to have achieved widespread use. Why a language based faithfully on (a) relational theory and (b) generally accepted principles of good language design might have been so much better.

askew, *adj.* at or to an oblique angle; awry
awry, *adj.* twisted to one side; distorted, crooked; wrong; perverse
wall, *n.* ...; a barrier; ...

"Askew Wall" is rather contrived as a pun but it works very well for me as a metaphor, as you can see from those dictionary definitions.

This lecture descends from one first given to a British Computer Society audience in December, 1987. In the early 1990s it found its way around various conferences and British universities, including Warwick. It was not devised with undergraduate courses in mind!

The Third Manifesto (*TTM* for short) is a paper by the author and C.J. Date, originally published in 1995, inspired by the many mistakes in SQL. It sets out a series of prescriptions that must be followed if a language is to be a faithful implementation of "the relational model of data" (as its inventor, E.F. Codd called it back in 1970). And a series of *pro*scriptions (things that must *not* be included) that are really consequences of the *pre*scriptions. And a series of Very Strong Suggestions. The paper is explained at length and in detail in a book, *Databases, Types, and The Relational Model* (Addison-Wesley, 2006) by the same authors.

My book *SQL: A Comparative Survey*, available at http://bookboon.com/en/textbooks/it-programming as a free download, examines SQL by comparing it, section by section, example by example, with relational theory and **Tutorial D** as taught in my earlier Bookboon publication, *An Introduction to Relational Database Theory.*

2. Terminological Equivalences (?)

I used to say that the terms in the left-hand column are synonymous with their counterparts on the right, and apologise for mixing them freely. But during our work on *TTM* Date and I realised that it would be much more sensible to reserve the terms *table, row,* and *column* for the SQL concepts and stick to the more formal terms when we really do mean relations and their properties. And under that distinction—SQL versus the relational model—they definitely are *not* synonymous.

3. The Perversity of SQL

The first SQL query on this slide was once set as an exercise for students of The Open University, who were asked to explain in ordinary English what the query is really asking for. Even the students who did understand SQL "subqueries" struggled with it, because of the perverse order in which the main "less than" comparison is written. And so did I, a tutor to some of these students. Anyway, the example set in motion a long sequence of events that was to culminate in a change to the SQL international standard (!) as I relate in the lecture.

Which is easier to understand: cities where 4 is greater than the number of other cities with bigger populations, or cities where the number of other cities with bigger populations is less than 4? Is a mile less than the longest distance you can run non-stop, or can you run for more than a mile, non-

stop? Is 100 less than the best score Shane Warne has ever made at batting in cricket? Or has Shane Warne ever scored more than 100?

It's only a psychological issue, of course, but the example illustrates one of the many examples of poor language design by the research team at IBM that invented SQL back in the 1970s: the revised query in the lower half of the slide was illegal in their language, even though it merely converts a couple of examples of $a > b$ into $b < a$. And the error still survives in many SQL implementations to this day, even though it was corrected in the international standard as long ago as 1992 ...

4. The Third Manifesto

... However, shortly after the publication of SQL:1992, a proposal was accepted to add to the Technical Corrigendum for SQL:1992 a clause to the effect that an implementation seeking conformance to just the "core" level of conformance would be permitted to keep that restriction on subqueries in comparisons. It was that proposal, approved at the committee meeting in Munich, Germany, in January 1993, that so exasperated the present author as to cause him to sit down in his hotel room with pencil and paper over one weekend, produce the first draft of *The Third Manifesto*, and fax it to Chris Date. (For what it's worth, that first draft is available in PDF at www.thethirdmanifesto.com.)

5. References

Apologies for referencing nothing but my own work and Chris Date's, but not a lot of people have written about this particular subject, SQL and the relational model, in this way.

6. A Brief History of Data

In the days of cards and tapes, all files had to be accessed sequentially, starting with the first record and then repeatedly moving on to the next one until the last one had been processed. This involved the coding of *loops* in application programs, *nested* loops when more than one file was needed , as was usually the case. The algorithms involved were quite tricky and a common source of *bugs* in the application programs.

The advent of disks allowed us to dispense with some of the loops but introduced the equally trap-laden process of *pointer chasing.* Disks also raised the possibility of making an organisation's data a *shared resource* for concurrent access by a possibly diverse community.

Codd recognized both the opportunity and the dangers. He sought a foundation for technology that would allow application programs working with what were later to be called *databases* to avoid both the use of loops and the use of pointers.

The line for 1975 mentions two languages, ISBL and SQL. ISBL was a brilliant piece of work done by a small research team in the United Kingdom. A notable omission from Codd's papers was any precise definition of a computer language that would materialize his idea. The ISBL team made good this omission, faithfully adhering to Codd's model. The System R team working in the USA had a slightly different objective: to show that Codd's idea could be implemented efficiently enough for commercial use even in the "on-line transaction processing" (OLTP) environment. But the System R team of skilled "engine-room" workers didn't seem to understand Codd's idea as well as the ISBL team and arguably weren't very good language designers either.

It isn't always the good work that achieves high visibility and acclaim.

The lines from 1990 onwards mention ideas that were the hot topics of the decades in question, sometimes portrayed at the time as spelling the end of the road for relational databases. I characterize "big data" as "back to square one" because it appears to transfer all the hard work of data manipulation, querying, integrity checking and so on back to the applications, just like it was in prerelational times! In a similar vein we have the "NoSQL" lobby. Well, I have a lot of sympathy

with the idea of abandoning SQL, but unlike those people I would abandon it in favour of *truly* relational database management systems!

7. A Brief History of Me

Terminal Business System was an early DBMS, before that term had even been coined. Working on it, as a fairly junior programmer to begin with, I learned about the problems of database management and the typical solutions that were known about in those days. But Terminal Business System applications had to use loops and chase pointers. We included a couple of simple scripting languages for maintaining the database and generating reports, but in each case a script could access no more than one file and required the file to have records of uniform format. These scripting languages were called, prosaically, File Maintenance and Report Writer. Users complained about their limitations. File Maintenance scripts couldn't include checks for consistency with other files in the database, or even among different records in the same file; and Report Writer scripts couldn't combine information from several files.

So that's why my attendance on Chris Date's course in 1972 was a "personal watershed". His account of the new idea from E.F. Codd gave us the ideal solution to the very problems our customers were begging us to solve. And I had been working, in vain, seeking possible solutions that very year. The real solution that now presented itself as far as Report Writer was concerned was beautifully simple: just replace the input file by a relational query result and leave everything else the same.

Six years later it was time to develop a new DBMS and by now the relational model had attracted enough attention to make it the obvious choice. We looked around the research world to see what it had made of Codd's idea and discovered ISBL, which not only solved all of the problems in language design that we had been grappling with but did so in a delightfully simple way, fully embracing the use of *attribute names*. We also discovered SQL but quickly rejected it as being not only unfaithful to the relational model but also extremely baroque and unconventional. And *relationally incomplete*. I assured the executive manager who was to approve our funding that SQL would never catch on!

8. Blank Slide

No notes. These "blank slides" (real ones in the lecture) are for some lecture moments that are intended to convey in a light-hearted way my very serious overall message.

9. Blank Slide

No notes.

10. Blank Slide

No notes.

11. Blank Slide

No notes.

12. Blank Slide

No notes.

13. The Wall Around Relationland

No notes.

14. What The Askew Wall Has Done

"Stifled research" is perhaps a little strong. But so much research work since 1980 has been compelled to address SQL rather than any decent relational language.

"Higgledy-piggledy" because OO takes us back to pointer-chasing (via oids), one of the very things the relational model seeks to avoid.

15. The Good Things The Askew Wall Has Done

The trouble is, what SQL means by a *table* is something that bears only a superficial resemblance to a *relation*. Examining the differences is the main purpose of this lecture.

"Query language ALMOST closed ..." because the tables that result for its use do not necessarily adhere to the rules governing tables that are stored in the database. We will examine those differences too.

16. The "Fatal Flaws" of SQL

This is just a list of the topics that follow. Each one represents a major departure from relational theory. The consequences of such departures need to be examined and understood, and their severity needs to be assessed. "Fatal flaws" was E.F. Codd's term for the most serious errors in SQL.

17. COLUMN NAMING FLAWS

In this section we examine the consequences of SQL's failure to adhere to the definition of *heading*, whereby each attribute of a relation has a unique name by which it can be referred to.

18. A Thematic Query Example

We examine the column-naming problems with reference to this example.

19. Anonymous Columns

The example on the slide is self-explanatory, but one interesting consequence of it is not mentioned here or later. The table resulting from the query cannot possibly be the value of any base table. That is because a base table is required to have a name for each column (and no two columns of the same name). Imagine a programming language that supported numerical expressions that cannot be assigned to a numerical variable!

I take the opportunity to introduce the terms *correctable* and *bypassable*.

An error in a computer language is correctable if it is possible to correct the error by a change to the language that will not affect the working of existing applications of that language.

An error in a computer language is bypassable if it can be avoided by judicious use of the language. "Judicious use" might entail the kind of use that experienced programmers would call a *hack*.

20. FROM Clause Restricted to Named Tables

Notice that it is essential to be able to assign a name to that "calculated column" if the query is to be used as a subquery in a FROM clause; otherwise the column cannot be referenced in the outer query.

The relational algebra was devised to enable queries of arbitrary complexity to be expressed in a conventional manner. Every expression can be written as an operand of an invocation of some relational operator to yield a bigger expression.

In the original SQL of 1979, an SQL query was not permitted to be an operand of a bigger SQL query unless it appeared as a "subquery" in a WHERE or HAVING clause. Not allowing it to appear as an operand in the FROM clause violated the very principle I have just described. As a result, original SQL was not even *relationally complete*.

If a language is relationally complete, we know that it can be used to express anything that can be expressed in first-order predicate logic, with certain well-understood restrictions on the use of

disjunction (OR) and negation (NOT) that turn out to be of no practical consequence. Thus we can be confident, when called upon to tackle some really complicated and difficult query or constraint, that there definitely *is* some solution.

If a language is not relationally complete we lack this confidence and so we might (a) spend much time looking for non-existent solutions and (b) give up in disgust on some problem that is actually solvable, believing it to be one of those insoluble problems that we know to exist. Indeed, the example on this slide is an excellent example of case (b). I used it for years before a friend who was a real expert showed me how to do this query in original SQL. I kept the example anyway because it nicely illustrates my point without invalidating it.

It's very easy to show a query that cannot possibly be done without using subqueries in the FROM clause. Suppose you needed to join together the results of two queries each of which uses a GROUP BY clause and aggregate operators in the SELECT clause. In the original SQL implementations there wasn't even the hack of creating a view for each of the GROUP BY queries and using the view names in a FROM clause, because of an arbitrary restriction that was placed on the use of views to prevent you from doing such a thing.

Okay, so this problem was fixed in the 1992 edition of the SQL international standard, but it still took years for most of the implementations to remove the restriction and to this day there are still many that haven't.

21. The FROM clause fix

The obvious fix was to use "AS" to give a name to the column. And that enabled the removal of the restriction on FROM clauses. But see how "clunky" the language becomes when it is put to the new tasks it is now able to take on.

22. With WITH

Oracle and IBM's DB2 both support WITH and I believe the restrictions I know about in DB2 apply in Oracle too. The main one is that WITH is permitted only on the outermost query and not on any subquery, wherever it appears.

23. Duplicate column names (1)

It might seem that the two columns actually have different names, E.Name and S.Name, but this is not the case. The qualifiers E and S are available only within the SELECT, WHERE, GROUP BY and HAVING clauses that appear at the same level as the FROM clause in which E and S are defined. Thus, the following expression is a syntax error:

```
SELECT S.Name FROM
        (SELECT * FROM Exam_Marks E, Student S WHERE E.Name = S.Name) AS T
```

and so is this:

```
SELECT T.Name FROM
        (SELECT * FROM Exam_Marks E, Student S WHERE E.Name = S.Name) AS T
```

In the first case the qualifier S used in the first SELECT clause is undefined because the scope of S is confined to within the parenthesized expression where it is defined. In the second case T is defined but T.Name is ambiguous because there are two columns called Name in the table whose rows are referenced by T.

24. Duplicate Column Names (2)

Isn't it crazy that SQL lets you write such an expression? Imagine a programming language that lets you declare two variables with the same name and having exactly the same scope. I suspect

that no such programming language exists. Say the name is X. Then any subsequent reference to X in that scope would be ambiguous. And indeed, the SQL standard makes

SELECT X FROM (SELECT Col1 AS X, Col2 AS X FROM T) AS dummy

a syntax error even though the expression inside the parens is legal. In other words, if you choose to give two or more columns the same name, that's fine, so long as you don't try to reference them. In fact, even

SELECT * FROM (SELECT Col1 AS X, Col2 AS X FROM T) AS NewT

is a syntax error, because * is defined to be shorthand for a list of the names of all the columns of NewT, which in this case is X, X.

25. A Fix for Duplicate Column Names (1)

As you might remember, NATURAL JOIN is spelled just JOIN in **Tutorial D**. It was added to SQL in 1992 but not all SQL implementations support it to this day.

Note that the only other way in SQL of selecting just one of the duplicate columns along with all the other columns is to write out all the column names, because SELECT * gives you both of the duplicate columns.

26. Why NATURAL JOIN is "Natural"

No notes.

27. DUPLICATE ROWS

Personally, I rate this phenomenon as the second-worst of all SQL's multifarious errors. The column-naming errors are more ridiculous but as *logical* errors I put them in third position behind duplicate rows. (NULL—see later—is the winner, by a large margin.)

28. Snark

Lewis Carroll's famous poem, written in the middle of the 19th century, pokes fun at the duplicate row phenomenon. Carroll (real name Charles Lutwidge Dodgson) was himself a logician and liked to have fun with logic. The poem starts like this:

> "Just the place for a snark!" the Bellman cried,
> As he landed his crew with care,
> Lifting each man on the tip of the tide
> By a finger entwined in his hair.
>
> "Just the place for a snark! I have said it twice.
> That alone should encourage the crew.
> Just the place for a snark! I have said it thrice.
> What I tell you three times is *true*."

In my lectures I am in the habit of repeating (just to make sure it is true), "There is no sense in which the same tuple can appear more than once in the same relation."

29. Duplicate Rows

The slide shows the three ways by which it is possible in SQL for the same row to appear more than once in the same table:

- declare no primary key or UNIQUE constraint for a base table;
- fail to write the word DISTINCT after SELECT in a query;
- write the word ALL after UNION in a query.

It follows that the phenomenon can be avoided by never doing any of these three things. But always writing DISTINCT is far too much of a pain, especially as it is likely to slow down execution of queries in which no duplicates would arise even when DISTINCT is omitted. (Conversely, you can sometimes speed up a UNION query by writing ALL, which is harmless when you happen to know that the UNION operands are guaranteed to be *disjoint* (have no rows in common).

30. Are Duplicate Rows Really Harmful?

Saying the same thing more than once, though it doesn't make it any truer than saying it just once, might be thought to be harmless. And much of the time it is indeed harmless. But it can be quite harmful when the duplicate rows appear in the final result presented to the user, and *very* harmful when they are included in the calculation of some *aggregation*—here they can result in *wrong answers*, as the example on the slide shows.

31. NULL

SQL's worst error! We do of course understand why the System R researchers invented NULL, but if only they had given the matter a little more thought they surely would have backed off from the idea.

32. NULL

The idea that a comparison should be neither true nor false when one of the comparands is NULL was inspired (?) by the observation that NULL would sometimes stand for "value exists but is unknown". In that case, how can we know whether it is less than, greater than, or equal to some given value? SQL's solution had the interesting (and disastrous) consequence that even NULL = NULL does not evaluate to TRUE. Besides, SQL users sometimes use NULL where the meaning "value exists but is unknown" does *not* apply. And so does SQL itself. For example, in SQL the sum of no numbers is NULL, whereas mathematically it should of course be zero.

If NULL appears somewhere where it does not mean "value exists but is unknown", SQL's treatment is clearly inappropriate and likely to give "wrong" results. Unfortunately E.F. Codd himself sought to rectify matters, not by dispensing with NULL (and the third truth value) altogether, but by proposing two different varieties of NULL and a *fourth* truth value.

These >2-valued logics lead us into very difficult waters indeed. My slides show you some of the strange effects and traps you get with SQL's treatment, but really they only scratch the surface. Classical logic comes with an assurance of *soundness* (if you can use it to prove something from some given assumptions , what you have proved is as true as those assumptions are) and *completeness* (if something is true, then it can be proved to be true, using the given *rules of inference*). We do not know if SQL's logic is sound or complete, because we don't even know what its rules of inference are (if any).

33. Is NULL a Value?

I claim not. For if v is a value, then it follows that $v = v$ is TRUE.

34. 3-Valued Logic: The Real Culprit

See the notes for Slide 32.

35. Surprises Caused by SQL's NULL

No notes.

36. Why NULL Hurts Even More Than It Once Did

In original SQL the only types that were permitted for columns were the various numeric types, CHARACTER types, DATE, and TIME. Values of such types are all considered to lack visible

structure, even though one might perceive of a character string as containing its constituent characters in a certain order, a date as consisting of a year, a month and a day number.

Thus, although NULL might appear in place of a value in a column of type CHARACTER, there was no such thing as a character string containing NULL in one or more positions, such as the character "A" followed by NULL for the second position, followed by "C" in the third position. Nor was there such a thing as a date with a given year and month but NULL for the day number. And those observations still hold, of course.

However ...

37. How *x*= *x Unknown* Yet *x* NOT NULL

The inventors of SQL back in the 1970s did not anticipate the need for user-defined types. Also, Codd's "first normal form" (1NF) was universally interpreted in those days as an outright prohibition on columns in which tables or rows are stored—i.e., columns of type TABLE(...) or type ROW(...). Much later it was realised that this interpretation of Codd's dictat was too strong, especially considering that aggregate operators like **Tutorial D**'s SUMMARIZE cannot be satisfactorily defined without the concept of relation-valued attributes.

During the 1990s the leading SQL implementations sought to make good these deficiencies but in many ways they were hamstrung by the errors of the past. Here we look at some of the unfortunate consequences of NULL in connection with types whose values have visible structure to them—values within values, one might say. A value of user-defined type POINT might be perceived as consisting of an X value and a Y value. A value of some ROW type consists of a value for each column of that type.

So, what has now happened is that a POINT "value" can exist that is not it itself NULL but nevertheless has NULL for its X component, or its Y component, or both.

A consequence of examples such as those shown on this slide is that a simple test for NULL doesn't always achieve what one would like to achieve. The user who defines a type such as POINT has to define and implement the operators for that type. A common decision taken by such developers in connection with NULL is simply not to support it, because of all the problems it causes. Therefore they like to place tests for NULL at the start of the implementation code, usually like this:

```
IF p1 IS NULL OR p2 IS NULL OR p3 IS NULL ...
THEN SIGNAL ERROR (...) ;
END IF
```

where p1, p2, p3, ... are parameter names. But those tests are inadequate in the case where an argument might contain an imbedded NULL without actually *being* NULL. Note that replacing "p1 IS NULL" by "NOT (p1 = p1)" doesn't help. Why not?

38. TABLE_DEE AND TABLE_DUM

Recall that TABLE_DEE and TABLE_DUM are pet names for the only two relations of degree zero—RELATION { TUPLE { } } and RELATION { } { }, respectively, in **Tutorial D**. Their existence and possible importance was noted by the inventors of ISBL at around the same time as IBM's System R team was working on SQL.

The SQL counterpart would be tables with no columns, but alas the inventors of SQL were unaware of their existence. So was Codd! When they were pointed out to him, his reaction was, "What possible use can they serve?" or words to that effect.

39. TABLE_DEE and TABLE_DUM

Recall also that in **Tutorial D** PER (TABLE_DEE) is used in an invocation of SUMMARIZE where no "grouping" is required—or rather (and better), the whole input relation is to be treated as a

single “group”. PER (TABLE_DEE) is **Tutorial D**’s counterpart of omitting the GROUP BY clause in SQL.

40. Failure to Recognise DEE and DUM

The consequences might appear to be rather minor, and so they are compared with the consequences of NULL, duplicate rows, and the various mistakes concerning column names. Nevertheless, it has been an inconvenience to researchers in “natural language query”, whose software is easily capable of processing queries of the form, “Are there any ...?”. Typically they first convert the human language sentence into a predicate that is then to be translated into SQL. Each free variable in the predicate becomes a column in the SELECT clause of the SQL query. But when there are no free variables the result would be a SELECT clause with nothing in it, which SQL does not support.

It is also interesting that various SQL implementations have proprietary extensions to get around the FROM clause problem. Some allow the FROM clause to be omitted, correctly returning a one-row table. Some use a special table name, such as Oracle’s curiously named DUAL (which on close examination turns out to be a bizarre beast, actually a table *variable*, having one column, named D and of type CHARACTER, and a single row with D = 'X').

41. Bypasses for Absence of DEE and DUM

Example 6 effectively extends TABLE_DEE to give RELATION { TUPLE { 'Yes!'} } if that is the result of the query. I was spent a great deal of time trying to express a query that would not only give RELATION { TUPLE { 'Yes!'} } when that is the answer but also give RELATION { TUPLE { 'No!'} } when *that* is the answer. Without using IF ..., that is.

42. It Could Have Been Worse ...

This slide just shows a curious effect that would have arisen with IS [NOT] NULL if SQL had recognized the existence of the 0-tuple. Recall that $\forall x \in A(\mathrm{P}x)$ is always true when A is the empty set.

43. MISCELLANEOUS FURTHER FLAWS

I draw your attention to various flaws that do not arise from a poor understanding of relational theory (or a misguided reluctance to adhere to it).

44. “=” Is Not “equals”

The idea that trailing blanks should be ignored in comparison of character strings was taken from another well-known, widely used, and much derided computer language: COBOL. Although the error is not in itself a *relational* error (so to speak), it does have very unfortunate implications for what are supposed to be relational operators.

45. Indeterminacy in SQL

Consider the first example on the slide

```
SELECT DISTINCT X FROM (VALUES 'x', 'x    ' ) AS T(X)
```

As the two rows here compare as equal in SQL, only one row must appear in the result. Which one? The SQL standard specifies that *any* row that compares equal with those two can be the result—it doesn't even have to be a row that appears in the input table! The standard also decrees that if the table input to a SELECT DISTINCT has any column of a CHARACTER type, then the result is *possible non-deterministic* and thereby subject to various restrictions such as the one in connection with constraints mentioned on the slide.

According to the SQL international standard, the implementation may choose *any* value that would compare equal to 'x'; in other words, the letter "x" followed by any number of blanks. This means that such expressions are *indeterminate* (also sometimes known as *non-deterministic*). Indeterminacy can cause a lot of problems and good computer languages avoid it. The standard also decrees that if the table input to a SELECT DISTINCT has any column of a CHARACTER type, then the result is *possible non-deterministic* and thereby subject to various restrictions such as the one in connection with constraints mentioned on the slide. SQL is rife with such restrictions. They undermine *orthogonality* (the principle by which independent language constructs do not interfere with each other).

46. Tables and Multisets of Rows

SQL uses the term *data type* to refer to those types whose values can appear in all the usual places except that they cannot appear as values of persistent variables in the database (i.e., base tables), nor can they be results of queries. Table types, by contrast, are the ones whose values can appear in the database and be the results of queries. One consequence of the fact that tables cannot appear as operands to operator invocations is that they cannot be subject to comparisons, using operators such as $\subseteq$ or =.

So SQL has two distinct type systems. As a result, none of SQL's types is what is sometimes referred to as *first class*. A type is first class if *any* typed construct in the language can be of that type. A good principle of computer language design states that all types should be first class.

One consequence of the fact that tables cannot appear as operands to operator invocations is that they cannot be subject to comparisons, using operators such as $\subseteq$ or =. That and various other deficiencies in SQL arising from table types not being first class eventually gave rise (in 2003) to the introduction of a parallel set of types—multiset types—in SQL's other type system.

47. SQL's TABLE Operator

The title should perhaps be SQL's Table Operators, plural. The reason why TABLE(TABLE(*t*) is not necessarily equal to *t* is that the result is not required to have the same column names as *t*.

48. Clunkiness of SELECT-FROM-WHERE

The point I am trying to make here is that if the inventors of SQL had realised at the outset that support for "derived tables in the FROM clause" would be essential, they might have abandoned their idiosyncratic SELECT ... FROM ... WHERE structure in favour of conventional syntax that gives the user the freedom to choose which operators to use in which order (as in ISBL, **Tutorial D**, and all normal programming languages).

49. A Murky Story

No notes.

50. Why "Maintenance Nightmare"

No notes.

51. "Nobody Ever Asked For It"

In 2005, a month before the meeting at which the SQL proposal was rejected, I asked an audience of about 90 Oracle users to raise their hands if they had ever wanted to do this. Hands went up all around the room.

52. The Folly of "Structured Queries"

In view of the "clunkiness" of the second expression on this slide, the answer to my question might actually be "yes". But I would add, "Would we even have had this SELECT ... FROM ... WHERE ... structure?" The next slide continues the theme ...

53. The Folly of Coercion

When an argument to an operator invocation is not of the same type as its corresponding parameter, some languages sometimes support an implicit conversion. For example, some SQL implementations allow you to write X = '1' even when X is of some numeric type. The DBMS attempts to interpret the character string as a number in normal decimal notation. If the attempt is successful, the operation is allowed; otherwise it results in a run-time exception. *This is not in general a good idea.*

The subquery in Slide 3, "The Perversity of SQL", because it is being compared with a number in the WHERE clause, is *coerced* by SQL from a table to a number—specifically, the number that is the value of the only column in the only row of the result of that subquery. The inventors of SQL rightly thought that would be very convenient, but that didn't make it a good idea.

In **Tutorial D** you would have to write CityCount FROM TUPLE FROM in front of the subquery, and the subquery would have to assign the attribute name CityCount to the result of COUNT. You might find that rather inconvenient, but at least it means that there is never any doubt about the type of an expression in **Tutorial D**, and that greatly helps the language to have the desired property of *orthogonality* (see the notes for Slide 45).

54. Why The Flaws Are "Fatal"

"Fatal" was Ted Codd's word, not mine.

- The Shackle of Compatibility.

  A bad construct cannot be removed from the language because existing applications might be using it.

- The Growth of Redundancy

  There's nothing intrinsically wrong with being able to solve the same problem in various different ways. But in a well-designed language redundancy is deliberately added when *shorthands* are devised for commonly needed operations that can already be expressed but whose expression is too complicated for comfort. Furthermore, those shorthands are in keeping with the existing style of the language.

  The relational operator COMPOSE, for example, was included in **Tutorial D** to illustrate this concept.

- Difficult or impossible to extend.

  One reason for the lack of support for temporal data in SQL might very well be to do with NULL and duplicate rows. If the same row appears more than once in a table and the user attaches some significance to those multiple appearances, how is that multiplicity to be preserved across a PACK operation? And what does PACK do if NULL is encountered in the interval column on which the packing is to be done?

- Also shackled by existing style.

  I once had a change proposal against the SQL standard rejected by the committee when some members argued that my proposed syntax was "too elegant"—not SQL's normal style! I was unable to argue against that point and so gracefully accepted defeat. I have completely forgotten what the proposal was about.

55. Errors Here to Stay

The Shackle of Compatibility means that even SQL's worst errors cannot be corrected.

56. The Growth of Redundancy

Which, if any, of these language constructs would you keep in SQL if it weren't for *The Shackle of Compatibility*—in other words, if we were allowed to discard features that are no longer needed?

Personally, I'd keep subqueries, in the interests of orthogonality and because they are sometimes convenient. I'd also keep GROUP BY. Note that **Tutorial D** has counterparts of both of these constructs.

57. OBJECT SUPPORT

Object-oriented programming hit the scene in the 1980s and towards the end of that decade much interest arose in the possibility of extending the paradigm to databases. As well as using OO concepts as a basis on which to design DBMSs, people thought about incorporating those concepts into existing database languages. Little work was done on incorporating them into *relational* database languages, however, because we didn't have any. We only had SQL as a nearest approximation to a relational database language, a situation that has continued (in the industry at least) to the present day.

Needless to say, there was indeed much interest in adding OO constructs to SQL, thus giving the DBMS some new opportunities for making mistakes, including very bad mistakes.

58. Object Oriented Databases

The first "good idea" is what we might take as *the* distinguishing feature between OO databases and relational databases. The mantra we were given in about 1990 was "persistence is orthogonal to type". In other words, whatever types are permitted for local variables in the OO programming language must also be permitted for *persistent* variables. This is an excellent idea for certain types of application, but it is not such a good idea for general use where ease of querying and declaring constraints is required. Relational databases are able to support ad hoc queries and declarative constraints precisely because of the uniformity of structure enforced by restricting the variables in the database (i.e., the persistent variables) to be, specifically, *relation* variables.

The "type inheritance" that is a feature of typical OO languages did not strike the authors of *The Third Manifesto* as conceptually agreeable. If T2 is a subtype of T1, meaning that every value of type T2 can be substituted wherever a value of type T1 is expected, then surely every value of type T2 *is a* value if type T1. But it seems very counterintuitive if that means that every point in 3-dimensional space *is a* point in 2-dimensional space. Instead, The Third Manifesto advocates subtyping *by constraint*. For example, every circle *is an* ellipse whose axes happen to be equal.

59. Rapprochement

From one of my dictionaries:

rapprochement *n.* a drawing together: establishment or renewal of cordial relations. [Fr.]

60. A Multimedium Relation

If you can forgive the feeble clip-art, this slide illustrates the idea of rapprochement, though I'm not sure we really needed Object Orientation to bring the ideas in question to Relationland. The heading of my bird table is assumed to be something like { Info XML, BirdName CHAR, Pic PICTURE, Video MOVIE, Song SOUND, Migr MAP } and the DBMS is expected either to support those types directly or permit its users to define them (and the operators that go with them).

User-defined type support is of course a central feature of object-oriented languages, though the term *class* is usually preferred to type and the rather inapt term *method* is for some reason preferred to operator.

All along relational theory has been independent of the types chosen for attributes of relations. But back in 1970 nobody was dreaming about things like pictures, videos, sound recordings and maps. In the spirit of keeping things simple, Codd proposed what he called First Normal Form (1NF) but his definition of this form was imprecise and gave rise to a great deal of confusion. His definition consisted of two points:

(a) In every tuple of every relation there shall be exactly one value for each attribute, drawn from the domain declared for that attribute. (Nowadays we call the domain a type, such as INTEGER.)

(b) The values that constitute a domain shall be "atomic".

It was point (b) that gave rise to the confusion, it not being clear exactly how Codd intended atomic values to be distinguished from nonatomic values. His attempted definition of atomic value was "[a value that] cannot be decomposed into smaller pieces by the DBMS (excluding certain special functions)". Even if we ignore the peculiar remark in parentheses here, the definition merely gives rise to further questions. What does "decompose" mean, precisely? And what is the significance of "by the DBMS"? Furthermore, it wasn't even clear whether the definition was intended to be a firm rule, meaning that the only relations to be supported at shall be 1NF ones, or merely database design advice as with all the other normal forms.

Notice, by the way, that part of the confusion arose because in those days the careful distinction between relation values and relation variables was not made at all in the database literature at large. Did Codd mean for all *relations* to be in 1NF (i.e., operators such as **Tutorial D**'s GROUP were not to be supported) or was he referring just to the *relvars* constituting the database?

Anyway, the upshot was that the first relational and SQL implementations all stuck to "atomic" types such as numbers, strings, dates and times. Thoughtful people did question the notion of atomicity, though. Can't a date be "decomposed" into a year, a month, and a day? Can't a string be decomposed into its individual characters? Can't an integer be decomposed (in a different sense of that word) into its prime factors?

Eventually it was agreed to regard point (b) of Codd's definition as misguided. As for point (a), that appears to be redundant because it is part of the definition of a relation and is a direct consequence of the notion of a relation as representing the extension of a predicate: when we *instantiate* a predicate we substitute a single designator (value) for each free variable.

Anyway, if we can agree to reject Codd's 1NF on the grounds that not only is its definition unclear but nobody has been able to replace it by something clear and agreeable, then we can claim that my bird table has *always* been supported by relational database theory. However, we might as well acknowledge the impetus created by the arrival of object orientation as pushing us in the right direction.

61. SQL for The Bird Table

The heading used here isn't *exactly* the one I suggested in the notes for Slide 60 but to all intents and purposes it's the same. The recent arrival of built-in type XML into the SQL standard and several implementations means that the first CREATE TYPE statement wouldn't be needed if the encyclopedia extract for each species was done in XML. One could also imagine the other "multimedia" types being built-in, using standard formats for images, videos and sound. But it makes—or rather, *should* make—no logical difference as far as the user is concerned whether any particular type is built-in or user-defined ...

62. The New Fatal Flaw

... but once again SQL flies in the face of reason.

The two blunders on display here are (a) to confuse types with relations and (b) to reintroduce pointers into the user's view of the data. Strangely, the First Great Blunder was being committed pretty well universally by speakers from Objectland at database conferences I attended around the turn of the 1990s.

The clause REF IS Birdref SYSTEM GENERATED defines a column named Birdref whose values are in fact pointers. The declared type of this column is called REF(Birdstuff).

63. Possible Reasons for The "Great Blunders"

It is perhaps unfortunate that the term *attribute* was used for what are also called the *instance variables* of a class. This perhaps lured Objectlanders into making that wrong equation. One might also point a finger at *entity-relationship modelling*, a discipline that lures database designers into regarding tuples as representing "entities" instead of what they actually do represent (propositions that are instantiations of predicates).

64. Object DB Structure

This slide just shows how pointers (in the form of oids) are used to connect things together in a object oriented system.

65. Relational DB Structure

And this one depicts the absence of pointers. There are no lines leading into a variable, and none leading out of one.

66. CLOSING REMARKS

No notes.

67. The Relationlander's Promise

I made this promise to myself in about 1980 and I have kept it ever since.

68. The Dream Database Language

No notes.

69. D

We use **D** as a generic name for any language that conforms to *The Third Manifesto*. Some people like to include **D** in the language name, as we did in **Tutorial D**. The term *Industrial D* is sometimes used for the one that doesn't exist (yet).

70. Projects based on *TTM*

One that isn't mentioned on the slide is D♭ (D-flat, to go with Microsoft's C#!)

71. Some Guiding Principles

No notes.

72. Logical Differences

Date and I were both rather struck by this saying of the famous 20th century philosopher Ludwig Wittgenstein when it was drawn to our attention by a friend of mine. It took us several years to track it down. It appears that Wittgenstein happened to say it in the presence of fellow philosopher Peter Geach, with whom my friend had been acquainted when he was a student in the 1960s.

73. Values and Variables

The distinction between the two concepts is absolutely clear and uncontroversial, but there is so much lazy writing about!—especially in our field. That said, it seems that the idea of using the term

relation variable for that very concept had not occurred to any writer before Chris Date introduced it in the early 1990s. Certainly Codd used the term relation for both values and variables, and to this day the SQL term table refers to either a "base table" (or updatable view) or any query result, even in as formal a document as the international standard tries to be.

74. Data Types and Relations

No notes.

75. Types are Not Tables

The analogy here appeals to the fact the values of a type can appear as attribute values in the tuples of a relation. Under our agreed interpretation of relations they thus represent the designators (nouns) that are to be substituted for the parameters of some predicate. Thus the tuples themselves represent sentences.

76. Conceptual Integrity

Brooks had been working for IBM as the manager in charge of the development of the huge, baroque operating system, prosaically called OS, that was used on the big IBM mainframes of the 1960s and 1970s. His famous book documenting his experiences and conclusions was entitled *The Mythical Man-month*. Note carefully the italicization of the word *the* in the quotation. I heard him give it the same emphasis orally when talking about conceptual integrity in an after-dinner speech I attended in 1995.

77. Reims Cathedral

This is the example used by Fred Brooks in that book, *The Mythical Man-month* (1975). The cathedral took perhaps 100 or more years to build, but the builders stuck slavishly to the original concept and this was the result.

78. Conceptual Integrity

You might as well see the entire speech, apart from some irrelevant introductory lines. It is one of my own personal favourites from Shakespeare's plays. From *Hamlet*, Act 1, Scene 3, Polonius saying goodbye and giving some parental advice to his son, Laertes, who is departing to France for a while:

> ... There, my blessing with thee!
> And these few precepts in thy memory
> Look thou character. Give thy thoughts no tongue,
> Nor any unproportion'd thought his act.
> Be thou familiar but by no means vulgar.
> Those friends thou hast, and their adoption tried,
> Grapple them to thy soul with hoops of steel;
> But do not dull thy palm with entertainment
> Of each new unhatch'd, unfledg'd courage.
> Beware of entrance to a quarrel; but, being in,
> Bear't that th'opposed may beware of thee.
> Give every man thy ear, but few thy voice;
> Take each man's censure, but reserve thy judgment.
> Costly thy habit as thy purse can buy,
> But not express'd in fancy; rich, not gaudy;
> For they in France of the best rank and station
> Are of a most select and generous choice in that.
> Neither a borrower nor a lender be;
> For loan oft loses both itself and friend,
> And borrowing dulls the edge of husbandry.
> **This above all, to thine own self be true,**
> **And it must follow as the night the day,**

Thou canst not then be false to any man.
Farwell; my blessing season this in thee!

I like to imagine Polonius, in those lines shown in bold, wagging his finger at a computer language, hence the small change I made to them.

79. Blank Slide

No notes.

80. The End

No notes.

End of Notes