Programming Languages (CS302 2007S)

Codd - A Relational Model for Large Shared Data Banks

Comments on:

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* **13**(6), June 1970.

At what stage is the normalization procedure carried out on a relation? Why is it not possible to enter data (or form relations) in a normal form as an initialization step?

Remember, this was theory and not yet practice. The key idea was that people might think naturally in terms of nested relations (because there was some of that kind of approach in the more implementation-oriented database designs), which made it important to show that you could do without nested relations. Typically, one writes non-nested relationships from the get-go.

I do not fully understand the join operation described in section 2.1.3. On the first paragraph of page 384, the author writes that the join of Ã¢ÂR with SÃ¢Â is different from join of Ã¢ÂS with RÃ¢Â. Why is this case?

Here's a simple one. Suppose R = { <1,2>, <1,3>, <1,4> } and S = { <5,1>, <6,1> }. Since no right value in R is a left value in S, the join of R and S is the empty set. In contrast, the natural join of S and R is { <5,2>, <5,3>, <5,3>, <6,2>, <6,3>, <6,4> }.

Also, how does the join in the paper relate to the different joins (left, right, inner, outer joins) found in MySQL?

I'll admit that I don't know the different joins in SQL.

This paper was written in 1970, how has the relational model evolved over time? Does this model still apply many of the implementations noted in the paper (e.g the method for dealing with inconsistency)?

The key aspects of the model don't seem to have changed significantly, although some subtleties have been introduced. I think we've found that normal form is much more natural than he'd predicted.

Lastly, the author argues against tree-structured files and network models. Does this mean that relational databases do not store data in tree structures or is he referring to the external presentation as opposed to internal representation of databases? I ask this because I think I learned that 2-4 trees are one of the most efficient ways to store data, I may be mistaken.

He is arguing about a clear distinction between the abstract interface and the underlying implementation. And, in some ways, 2-3 trees (and, I suppose, 2-4 trees, although I know little about them) are appropriate for some tasks, but not every tasks. For example, I expect that a tree representation makes joins difficult.

I must confess that for most of the reading for tonight, I had no idea what he's talking about. I am interested in how the previous data bank models worked and what was so disadvantageous about them.

The previous models focused much more on the underlying structure. If you think about how you might arrange a collection of data without using a DBMS, you're on your way toward some of those underlying models. The disadvantages were the obvious ones - if you knew enough about the implementation, people tended to program to it, so if the implementation changed, they ended up with either non-working or badly-working code. Also, some model choices ended up working well for some kinds of data and less well for other kinds. The abstraction that Codd emphasizes makes it easier to have the implementation adapt to the particular data or queries.

I am also unsure as to the relationship between the sublanguage and the expressible set. Are they the same thing? Or does the sublanguage describe the expressible set?

The sublanguage is the language used (in Codd's apparent intent, as a kind of library for another language) to describe the relations. The expressible set is the set of relations describable by the sublanguage.

Also, what, exactly, does codd mean when he says that the universality of the data sublanguage lies in its descriptive ability, not its computing ability?

Abstraction, not implementation. His goal is to describe data, not to build a model of computation (which one would then be expected to prove equivalent to Turing Machines).

It seems that some of the ideas presented in this paper are also found in Object-Oriented programming. There seem to be parallels between the idea of data encapsulation and interfaces, to reduce implementation dependencies, and the idea of using a relation view of data to avoid ordering dependencies, access path dependencies and other implementation dependencies. Did the Object-Oriented paradigm influence the Declarative or was it the other way around? Or were both views simply applying the same theoretical abstractions?

I think both are applying the same general principle - abstraction is your friend.

The paper says normalization is an advantage for storage (381, third paragraph). I don't know how to prove this to be true. From the employee example, I can see the primary key man# are duplicated many times. Duplication doesn't sound like an advantage for storage.

The keys are typicaly integers, which doesn't add much overhead. I don't know enough about the previous designs to tell you how much overhead there was in making subrelations as values.

Is there a systematic method to design a relation or database tables? Like how to decide what and how many tables, primary key, foreign key are needed?

There are some clear ways to analyze problems to determine potential relations, but I expect that there is also some art (and therefore some experience) required.

Can you use a real problem to walk us through the process?

The paper walks you through a real example. If there's time on Friday, I'll try to walk you through another.

In the reading on page 379 in the bottom right-hand side paragraph, Codd first explains why in a relation, if columns are labeled by the name of corresponding domains, the ordering of columns matter. He gives an example of columns with the same name and why it is important there. However, next Codd says that several information systems fail to provide data representations for relations which have two or more identical domains. Why is that the case and here is he referring to the case where the columns are unordered?

My guess is that the designers did not see natural problems that required multiple domains. The ordering does not seem all that relevant.

In what kinds of systems would the three kinds of data dependencies i.e.ordering, indexing, access path dependence be similar or the same and how as Codd mentions on page 377 in the second last paragraph are the not clearly seperable sometims?

The three are different implementation techniques, They can be tied together, in that the implementation uses more than one, and they are interrelate.

What is the difference between a composition and a permutation? Aren't they all just projections?

A composition is a projection of a join. A permutation is a projection in which every column is preserved. So yes, they are both projections, but they are different kinds of projections.

How are relationships between tables defined in SQL? I have worked with MySQL but used PHP to deal with the relationships/data dependencies etc.

I'm not sure what you mean by "relationships". Typically, you indicate that one relation may be used in conjunction with another relation through foreign keys.

Can Codd's Relational Model fit under a language paradigm outside of declarative? It seems as though his ideas on "maximal independence between programs on the one hand and machine representation and organization of data on the other" fit perfectly under the declarative paradigm. It almost feels like Codd is creating a declarative language.

I think Codd was thinking about this relational model more in an imperative context (which is why he expresses it as a "sublanguage". But yes, many computer scientists think of this as the creation of a declarative language.

Looking at the date of the article, it seems like a radical idea for 1970 when computers were slow and data probably wasnÃ¢Ât stored in memory. Codd even seems like he is softening the blow at points when referring to the need for data independence as something needed in the "future".

I expect that “future” meant “near future”. And yes, it took some time for his ideas to be accepted.

With the exception of the "indexing dependent" database, I don’t understand how users would access things in these ’older’ styles of database. The paper seems to gloss over that and assume we’re already familiar.

It depends on the particular implementation, but, as I understand it, there was probably a decent amount of pointer chasing.

Because I don’t know how users accessed these old databases, I don’t really understand how the relational logic databases differ. The first descriptions Codd offers seems to emphasize more the underlaying implementation (and some of the information that could be accessed by using knowledge of the implementation), whereas the rest of the paper focuses on theory. But since Codd doesn’t give solid details for the implementation of this database, I don’t see how it necessarily differs from the first examples. It seems that the top examples, with little to no additional abstraction, could be used in this relational way. That, or maybe I can’t read.

You are correct. What he is suggesting is primarily that we need to separate more clearly the interface from the implementation.