

Formal Semantics for Production Databases: Part 1. Of Silos and Squiggles.

Dr. Tom Johnston
MindfulData.com
Jan 25, 2008.

In conferences, in conversations with fellow IT professionals, and in my own consulting experience, I have observed that ontologies and taxonomies are generally considered to be concepts associated with managing unstructured data. The IT projects that are introducing these concepts are “content management” or “document management” projects. They are definitely *not* projects to develop or upgrade production databases.

These projects thus introduce the risk of repeating the “silo mistake”, but on a far larger scale than ever. For if these projects result in software that can search for information on customers, and if they define “customer” as their subject matter experts, or as industry standards, define “customer”, then what of the definition of “customer” that describes our production databases? Suppose a search, using the ontology-based “inference engine” of a new document management system, finds documents about customers.

But in our production databases, a customer is – whatever the data dictionary may say – someone represented by a row in the Customer table. There are rules for when such rows may be inserted, and when they may be deleted. These rules are really what “customer” means on the production database side of the house.

Without policies, procedures and a lot of hard work, our enterprises are likely to end up with apples and oranges, with one meaning for its core concepts used to manage unstructured data, and a different meaning used to manage its structured data. And when the senior vice-president for sales looks at a sales report of selected customers, and then looks over at a legal contract governing our relationship with our customers, he is likely to assume that the contract applies to the customers on that list. But unless we have actively made the concept of “customer” identical across structured and unstructured data stores, the senior vice-president’s assumption is, at best, a rash one.

The “apples and oranges” problem, however, has the potential for delivering mis-information to business users across other fields as well. As supply-chain participants more tightly integrate their databases in order to streamline the pipeline, the opportunities for apples and oranges mis-information multiplies.

This is the risk which the new technology of ontologies, taxonomies and inference engines brings with it. As our databases are more and more updated and queried as part of a group of databases, we must insure that the objects of our updates and queries are the same. Interoperability as a technical function – as the ability to query across physical separate databases and locations – is effectively an accomplished fact. But the need I am pointing to is the need to guarantee *semantic* interoperability, the ability to access multiple databases in a single query or update where the data objects they refer to are not apples and oranges, where those data objects *mean* the same thing.

“Semantics”, “meaning” and their cognates are the language in which these critical requirements are now being stated. As I hope these articles will illustrate, they are a better, more nuanced, language than the old language of “information” vs. “data”. We have always managed data in order to provide useful information. But the newer technologies now on the horizon are going to make that job far more difficult than it has ever been before. For this more difficult job, this more nuanced way of stating requirements is appropriate and, indeed, even necessary.

“Semantics” and “Formalized Semantics”.

By “semantics”, I mean everything that allows someone to look at data – let’s say from squiggles on a screen – and understand the information being expressed by those squiggles. [For a brief discussion of what the semantics of data is, see my unpublished essay]

Ontologies and taxonomies – and associated concepts such as thesauri and controlled vocabularies – are a new and evolving set of data structures and processes whose purpose is to extend the ability of software to help us reason, i.e. to find information we are interested in, and to understand the data by which that information is expressed. As such, they are tools to

extend our ability to formalize the semantics of our database systems and associated business applications.

Now for “formalized semantics”. By this, I mean semantics which is expressed in a way that software can make use of it. The easy way to express semantics is to write it out in English. We do that when we write glossaries, and also when we write up the definitions of the entities, attributes and relationships of a data model. This is easy because spoken and written English (or any other natural language) is the very semantic air we breathe. When we write carefully-crafted definitions, the odds are good that those who read those definitions will understand us; they will “get our meaning”.

But one of the greatest inventions of all time, once we learned how to record meaning in squiggles, was the ability to develop rules to transform strings of squiggles into new strings of squiggles in such a way that those new strings of squiggles would represent new true statements, often new true statements that we didn’t realize were true before the squiggle transformation was applied.

Squiggle Transformations.

It will require later articles in this series to fully explain what I mean here. But those who have had even a freshman level course in logic will remember how they translated English arguments into statements of propositional or predicate logic, and then used formal rules of inference to attempt to derive the conclusion of that argument from all the other statements in it. We may have “understood” what those formal rules “meant”, e.g. that we can substitute $[\text{NOT-P OR Q}]$ for $[P \rightarrow Q]$, or vice versa, wherever they occur, that we can substitute $\text{NOT}[P \text{ AND NOT-Q}]$ for $[\text{NOT-P OR Q}]$, or vice versa, wherever they occur, that we can write $[Q]$ as a line in a proof whenever we have $[P]$ and $[P \rightarrow Q]$ as prior lines, and so on.

But the fact that we have software for proving theorems in logic makes it clear that understanding is not required to do the substitution, to translate given squiggles into new ones. What we know as the “rule of material implication”, or a “deMorgan’s transformation rule” can be implemented in software as code to translate one text string into another. And just as legal rules are designed (with the intent, at least) of occasionally letting the guilty go free in order to avoid the risk of mistakenly punishing the innocent, so

too rules of formal (truth-valued) logic are designed to occasionally let valid arguments lie beyond their ability to prove them, in order to avoid the risk of ever generating false conclusions from true premises.

Relational DBMSs, Ontologies and Squiggle Transformations.

Note: the use of the term “squiggle” here is not intended to be cute. It is intended to be *startling*. It is intended to shock us into realizing the almost miraculous power of formal mechanisms to generate new truths from old ones, thus increasing our fund of knowledge.

Theorem-proving software translates strings into new strings, nothing more. No meaning, no information, no semantics, no truth; just strings, strings and more strings.

Relational DBMSs as Theorem Proving Squiggle Transformers.

Relational DBMSs require us to store data in certain well-formed patterns, called schemas. The rules of the SQL Data Definition Language (DDL) implemented by a given RDBMS are the grammatical rules which define which patterns are well-formed.

So in a relational database, our data is recorded as instances of these schemas. A table defines the syntax of a statement about whatever kind of object the table represents (customers, products, invoices, etc). Each row in the table is a grammatically correct statement about one instance of that kind of object, about one customer, one product, and so on.

SQL Insert, Update and Delete transactions are commands of the SQL Data Modification Language (DML) to create, modify or delete one or more statements in the database, together with the data that is needed to successfully carry out the command. For example, a complete statement is needed to carry out an insert, but only a primary key is needed to carry out a delete.

Queries also statements of SQL DML. They are requests to find one or more rows which meet stated criteria, for each one to possibly combine it with other rows to create a single result row, to drop zero or more columns from

the resulting combined rows, and then to return all the found and modified rows to the query author, in a “result set”.

All this is probably obvious to most readers. So what is the point, other than to state the obvious in a perhaps novel manner? The point is to emphasize the continuity of relational theory and technology with the newer concepts and technologies of ontologies and their ilk.

For the query management piece of an RDBMS is a restricted form of theorem-proving software, focused on finding instances that satisfy statements of the form “There exists an X such that “. Thus, a RDBMS is an “inference engine”, a term we will come across again and again as we continue this discussion. But it is an inference engine fine-tuned to prove one particular type of theorem in formal logic. It is designed to prove the truth of existentially-quantified statements in first-order predicate logic. It proves them by finding instances of those statements in the database. It disproves them by finding no such instances. We query “Are there any customers in the South-East sales region, who have ninety days or more overdue balances of \$50,000 or more? And our theorem-prover goes to work.

Ontologies as Theorem Proving Squiggle Transformers.

{1/25/08. I’ll finish this section soon. That will conclude Part 1, and then we move on. }