Databases are Categories II
Refinements and Extensions

David I. Spivak

dspivak@math.mit.edu
Mathematics Department
Massachusetts Institute of Technology

Presented on 2010/10/22
On June 3, 2010 I gave a talk here at Galois called “Databases are categories.”

The main idea was that:
- a database schema is a category $C$, and
- a state on that schema is a functor $\delta : C \to \text{Set}$.

Here is a quick review:
Review of basic category theory

- A category $\mathcal{C}$ is a system of objects and arrows, and a composition law.
- A functor $\mathcal{C} \to \mathcal{D}$ is a mapping that preserves these structures.
- $\textbf{Set}$ is the category whose objects are sets, whose arrows are (total) functions, and whose composition law is the specification of how functions compose.
- If $\mathcal{C}$ is the category on the left below, then a functor $\delta : \mathcal{C} \to \textbf{Set}$ might look like this:

$$
\mathcal{C} := \begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow g & & \downarrow h \\
C & & 
\end{array}; \quad \delta = \begin{array}{cccc}
\bullet a_1 & \bullet a_2 & \bullet a_3 \\
\downarrow & & \downarrow \\
\bullet c_1 & & 
\end{array} \xrightarrow{(b_1, b_1, b_2)} \begin{array}{c}
\bullet b_1 \bullet b_2 
\end{array}
$$
How databases fit in

A category $\mathcal{C}$ as a schema: Each object $A \in \text{Ob}(\mathcal{C})$ is a table, each arrow $A \rightarrow B$ is a foreign key column of table $A$ pointing to table $B$. 

<table>
<thead>
<tr>
<th>Id</th>
<th>First</th>
<th>Last</th>
<th>Mgr</th>
<th>Dpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>David</td>
<td>Hilbert</td>
<td>103</td>
<td>q10</td>
</tr>
<tr>
<td>102</td>
<td>Bertrand</td>
<td>Russell</td>
<td>102</td>
<td>x02</td>
</tr>
<tr>
<td>103</td>
<td>Alan</td>
<td>Turing</td>
<td>103</td>
<td>q10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Secr'y</th>
</tr>
</thead>
<tbody>
<tr>
<td>q10</td>
<td>Sales</td>
<td>101</td>
</tr>
<tr>
<td>x02</td>
<td>Production</td>
<td>102</td>
</tr>
</tbody>
</table>

$\mathcal{C} = \left\langle \begin{array}{c}
\text{Employee} \\
\text{Department}
\end{array} \right\rangle$

$\delta: \mathcal{C} \rightarrow \text{Set}$

David I. Spivak (MIT)
The purpose of this talk is:

- to refine the above “databases are categories” notion a bit:
  - a database schema is really “a sketch.”
    In other words, a database schema is a presentation of a category.
  - We can use a similar idea to discuss incomplete data.

- to discuss querying and data migration.

- to discuss typing and calculated fields.

- to talk about ontologies, or “ologs,” and a vision of “functorial communication” based on the above ideas.
Question: So, what’s wrong with saying that the schema for the database on the left is the category on the right?

Answer:

- Categories have a composition law defined for every pair of composable arrows.
- But in the picture on the right, we don’t have an arrow for $s; f$.
- Similarly in the schema on the left, there is no “secretary’s first name” column in the department table.
- Conclusion: we are only imposing a composition law where we need to.
Why venture into sketches?

- We want to be able to write down a set of objects, arrows, and composition laws, without having to throw in an arrow for every possible composition. How do we encode that?
- We want to present categories by generators (objects and arrows) and relations (composition laws).
- We want to “sketch a category.”
Suppose we don’t have any composition laws to speak of. What information do we need in order to specify such a schema?

Information schema:

\[ G := \begin{array}{cc}
\text{Table} & \text{Column} \\
\text{foreign_table} & \end{array} \]

Here’s an example state on this schema, a functor \( \delta : G \rightarrow \textbf{Set} \):

<table>
<thead>
<tr>
<th>Object</th>
<th>Arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Id} )</td>
<td>( \text{Id} )</td>
</tr>
<tr>
<td>1</td>
<td>v</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
</tr>
<tr>
<td>2</td>
<td>z</td>
</tr>
</tbody>
</table>
Information schema for category sketches

Schema for category sketches:

Here we would need three additional equations coming from the triangle

\[
\Delta := \begin{array}{c}
\text{Object} \\
\text{target} \\
\text{source} \\
\text{Arrow} \\
\text{composite} \\
\text{second_map} \\
\text{first_map} \\
\text{Comm. triangle}
\end{array}
\]

\[
\text{first_map};\text{source} = \text{composite};\text{source} \\
\text{first_map};\text{target} = \text{second_map};\text{source} \\
\text{second_map};\text{target} = \text{composite};\text{target}
\]
Sketching our example, $C$

- $C$ is a schema for a department store.
- But how do we record the schema $C$ itself, in terms of the information schema $\Delta$?

\[ C = \]

\[ \Delta = \]

<table>
<thead>
<tr>
<th>Arrow</th>
<th>Id</th>
<th>source</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>d</td>
<td>E</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>s</td>
<td>D</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>f</td>
<td>E</td>
<td>E</td>
<td>S</td>
</tr>
<tr>
<td>l</td>
<td>E</td>
<td>E</td>
<td>S</td>
</tr>
<tr>
<td>n</td>
<td>D</td>
<td>D</td>
<td>S</td>
</tr>
<tr>
<td>i</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comm. triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>
We can write any schema $C$ as a database state on $\Delta$.
We refine our definition from last time to say that a database schema is a state on $\Delta$.
I’ll call these “pre-categories.” They are basically equivalent to categories.
A functor between pre-categories is just a morphism of states on $\Delta$.
I swept the difference under the rug last time because I wanted to emphasize the tight connection between database schemas and categories. That connection is still tight.
Now that we’re working with presentations of categories rather than categories,
it might be nice to “present” other facts, besides composition laws.
We can use more complex information schemas (beef up $\Delta$) to specify that a certain table in our schema $\mathcal{C}$
- is a product of other tables,
- is a fiber product,
- is a colimit,
- is an exponential object,
- is empty (i.e. has no rows),
- has only one row,
- etc.

We can do all this with sketches.
“Sketch” is category-theory terminology for “category specification.”

In a sketch we can specify that a certain object must be the limit or the colimit of some diagram.

This could be used, e.g., in a database where we want to have a table of “airplane seats” which is the coproduct of the tables “first class seats” and “economy class seats”.

We specify that we want \( S = F \amalg E \) in the sketch \( C \).

Now, instead of states being functors \( C \to \textbf{Set} \), states are “sketch maps” \( \delta : C \to \textbf{Set} \); i.e. functors that preserve all the specified facts.

For example for \( \delta : C \to \textbf{Set} \) to be a sketch map, we must have

\[
\delta(S) = \delta(F) \amalg \delta(E).
\]
At this point, one recognizes that sketches are quite similar to UML (Unified Modeling Language) diagrams for database schemas.

You just specify what you want.

What’s new here is the connection between database theory and category theory.

Category theory brings formal reasoning to the picture that was already there.
Sketching states with “RDF triple stores”

- Recall from the first talk that given a category $\mathcal{C}$ and a functor $\delta : \mathcal{C} \to \textbf{Set}$ one can take the Grothendieck construction $\text{Gr}(\delta)$.
- Suppose given the following example, considered as $\delta : \mathcal{C} \to \textbf{Set}$

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emp_Id</td>
<td>First</td>
<td>Name</td>
</tr>
<tr>
<td>101</td>
<td>David</td>
<td>Sales</td>
</tr>
<tr>
<td>102</td>
<td>Bertrand</td>
<td>Production</td>
</tr>
<tr>
<td>103</td>
<td>Alan</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Last</th>
<th>Secr’y</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Hilbert</td>
<td>101</td>
</tr>
<tr>
<td>102</td>
<td>Russell</td>
<td>102</td>
</tr>
<tr>
<td>103</td>
<td>Turing</td>
<td>q10</td>
</tr>
</tbody>
</table>

Applying the Grothendieck construction, we get a category $\text{Gr}(\delta)$:
Change of perspective

Given $\delta : \mathcal{C} \to \textbf{Set}$, the Grothendieck construction of $\delta$ gives a functor

$$\pi : \text{Gr}(\delta) \to \mathcal{C}.$$ 

The fiber over (inverse image of) every object $X \in \mathcal{C}$ is a set of objects in $\pi^{-1}(X) \in \text{Gr}(\delta)$. That set is $\delta(X)$. 
Allowing for incomplete, non-atomic, or bad data

- We can think of any functor $\pi : D \to C$ as a “pre-state” on $C$.
- Such a functor $\pi$ can encode incomplete, non-atomic, or bad data.

Row 103 has no data in the $f$ cell, and row 104 has too much.
- Bad data (data not conforming to declared composition laws) can also be encoded as a functor $\pi : D \to C$.
- Any pre-state on $C$ can be “corrected” in a canonical way to a state.
- Conclusion: we can use category theory as a model even when things are awry.
Let’s go back to the simple picture of database schemas as categories or pre-categories $\mathcal{C}$ and states as functors $\delta: \mathcal{C} \to \text{Set}$. (Not sketchy).

Given a schema $\mathcal{C}$, the category of states on $\mathcal{C}$ is denoted $\mathcal{C}–\text{Set}$.

Given a morphism between schemas, we want to be able to move data back and forth in canonical ways.

That means, given $F: \mathcal{C} \to \mathcal{D}$ we want functors

$$\mathcal{C}–\text{Set} \to \mathcal{D}–\text{Set}$$

and

$$\mathcal{D}–\text{Set} \to \mathcal{C}–\text{Set}.$$
The “easy” migration functor, $\Delta$

- Given a morphism of schemas (i.e. a functor)

$$F : \mathcal{C} \rightarrow \mathcal{D},$$

we can transform states on $\mathcal{D}$ to states on $\mathcal{C}$ as follows:

$$(\delta : \mathcal{D} \rightarrow \textbf{Set}) \mapsto ((\delta \circ F) : \mathcal{C} \rightarrow \textbf{Set})$$

- Thus we have a functor $\Delta_F : \mathcal{D}-\textbf{Set} \rightarrow \mathcal{C}-\textbf{Set}$.

- $\Delta_F$ basically operates by “re-indexing.” Using it, one can duplicate or drop tables or columns.
Given a morphism of schemas (i.e. a functor) $F : C \to D$,

- the functor $\Delta_F : D\text{-Set} \to C\text{-Set}$ has two adjoints:
  - a left adjoint $\Sigma_F : C\text{-Set} \to D\text{-Set}$, and
  - a right adjoint $\Pi_F : C\text{-Set} \to D\text{-Set}$,

$$C\text{-Set} \xleftarrow{\Sigma_F} D\text{-Set} \xrightarrow{\Delta_F} C\text{-Set} \xleftarrow{\Pi_F} D\text{-Set}$$

- Thus, given a morphism $F$ of schemas, these three functors, $\Delta_F, \Sigma_F, \text{ and } \Pi_F$

allow one to move data back and forth between $C$ and $D$ in canonical ways.
Views as polynomial functors

These functors can be arbitrarily composed to create views.

Given a state $\gamma: B \to \textbf{Set}$, what is $\Pi_H \circ \Sigma_G \circ \Delta_F(\gamma): \mathcal{E} \to \textbf{Set}$?
A simple “SELECT” query

SELECT title, isbn
FROM book
WHERE price > 100

\[ \Delta_G \circ \Pi_F \text{ is the appropriate view.} \]

For any \( \delta : C \to \text{Set} \), we materialize the view as \( \Delta_G \circ \Pi_F(\delta) \).
We are often interested in taking data from one enterprise model $\mathcal{C}$ and transferring it to another enterprise model $\mathcal{D}$.

Such transfers can also be accomplished using polynomial functors.

However, if $\mathcal{C}$ and $\mathcal{D}$ are not basic schemas (they’re too “sketchy”) then the “harder” migration functors, $\Sigma$ and $\Pi$, might not exist.

Also, we might need to perform calculations such as concatenation, addition, comparison, conversion of units, etc. in order to interface these schemas.

For this we’ll need an underlying typing category.
Incorporating data types and functions

In the example:

\[ C := \]

- \( \bullet_{\text{String}} \)
- \( \bullet_{\mathbb{R}} \)
- \( \bullet_{\mathbb{R}^{>100}} \)

how do we know that \( \bullet_{\text{String}}, \bullet_{\mathbb{R}}, \) and \( \bullet_{\mathbb{R}^{>100}} \) are what they say they are?

That is, given \( \delta: C \rightarrow \text{Set} \), how do we specify that \( \delta(\bullet_{\mathbb{R}}) \in \text{Set} \) is some pre-defined data type like Float.
Let **Hask** denote a category of types and functions that has been implemented on a computer and for which (at least theoretically) there exists a functor $\mathcal{V} : \textbf{Hask} \rightarrow \textbf{Set}$.

- Think of **Hask** as all Haskell data types and the definable functions between them, as well as all new types that could possibly be output by modules.

- Now **Hask** begins to look like a schema and $\mathcal{V}$ a “canonical state.”

- Since database schemas are categories and **Hask** is a category, we can integrate the two.
Example

- Here a nice functor

\[ \mathcal{B} := \begin{array}{c}
\bullet_{\text{String}} \\
\bullet_{\mathbb{R}^{>100}} \rightarrow \bullet_{\mathbb{R}}
\end{array} \]

\[ F(\bullet_{\text{String}}) = \text{char}(40), \quad F(\bullet_{\mathbb{R}}) = \text{Float}, \quad F(\bullet_{\mathbb{R}^{>100}}) = \text{some new type}. \]

- There is also an obvious functor

\[ \mathcal{B} = \begin{array}{c}
\bullet_{\text{String}} \\
\bullet_{\mathbb{R}^{>100}} \rightarrow \bullet_{\mathbb{R}}
\end{array} \]

\[ G \rightarrow \begin{array}{c}
\text{book} \rightarrow \bullet_{\text{String}} \\
\text{price} \rightarrow \text{isbn} \rightarrow \bullet_{\text{isbn-num}}
\end{array} = \mathcal{C} \]

- We are interested in functors \( \delta : \mathcal{C} \rightarrow \textbf{Set} \) equipped with a map \( \Delta_G \delta \rightarrow \Delta_F V \).
Typing in general

- If we need to enforce data types, our schema $\mathcal{C}$ will more than just a category (or sketch),
- It will be a category (or sketch) plus something like above:

$$\text{Hask} \xrightarrow{F} \mathcal{B} \xleftarrow{G} \mathcal{C}.$$ 

- And we won’t interested in any old state $\delta: \mathcal{C} \to \text{Set}$ but only those with a map $\Delta_G \delta \to \Delta_F V$, where $V: \text{Hask} \to \text{Set}$ is as above.
- By definition of adjunction, that’s just

$$\delta \to \Pi_G \Delta_F(V),$$

and $\Pi_G \Delta_F(V)$ is some huge fixed state on $\mathcal{C}$ that encodes our typing requirements.
- So the category of typed states is $\mathcal{C}-\text{Set}/\Pi_G \Delta_F(V)$. This is a topos.
Summary

- The longer portion of this talk is over.
- I discussed how to tighten the connection between databases and mathematics.
  - How to only include the columns and composition laws you want (not necessarily all of them).
  - How to force tables to be products or coproducts (etc.) of other tables.
  - How to model incomplete, non-atomic, or bad data.
  - How functors between schemas give rise to views and data migration.
  - How to encode typing information and calculated fields to database schemas.
In this short second section, I want to look at these categories and sketches from a different perspective.

Issue: how to allow people (e.g. scientists) to record very precise conceptual ideas.

- A big problem in science is to know where data comes from and how its been manipulated to arrive at conclusions.
- Another is to have the flexibility to design new databases on the fly, so as to capture unexpected nuances of data.
- Finally, academic researchers need to be able to record precisely what they mean in a computer-searchable way, rather than in silos of prose.

We need to democratize information storage.
The word “Olog”

- From “ontology log,” a play on “Blog.”
- Alternatively, a suffix for “study of.”

An olog is a category or sketch, but with natural language labels.

The goal is to capture meaning using explicit structure.
“Arginine is an amino acid. An amino acid has a side chain, a carboxylic acid, and an amine group. An electrically-charged side chain is a side chain. Arginine’s side chain is electrically-charged.”
Another example

(1,2,4,5) and (2,3,5,6) are fiber products.

1. a material $x$ at room temperature

2. room temperature

3. $\{x \in \mathbb{R} : 293 \leq x \leq 298\}$

4. $x$ is at room temperature

5. a material $x$ at a temperature $y$

6. a temperature

7. $y$ is a temperature in kelvin

- This is clearly a sketch defining room temperature and what it means to be a material at room temperature.
- An underlying typing system could handle the arrow $3 \rightarrow 6$.
- This olog could be referenced and extended by many scientists.
- It represents a fragment of a world-view.
- Moreover, each object and arrow could be “clickable” meaning one could open it up as a table of values.
Another example

1. a company that has hosted me

2. Galois Inc.

3. a company in Portland Oregon

4. a company

5. Portland Oregon

6. (Galois, John Launchbury)

7. a pair \((x, y)\) where \(x\) is a company and \(y\) is its founder

8. a founder

9. a person

10. a person I play go with

11. John Launchbury

David I. Spivak (MIT)
Imagine a world of people authoring ologs and connecting to each other’s ologs, forming a network of knowledge.

These connections could given by functors (or “spans”), which could only be instantiated if the two ologs are compatible.

My dream is an “olog network” of people recording their world-views.

- This is analogous to the semantic web vision (in fact it is possible to convert an olog into a RDF or OWL schema).
- Politicians could record their views in ologs. It would be interesting to note precise differences between Obama’s olog and Palin’s olog.
- Information in the olog network is neither right nor wrong, just “linked into” by others or not.

Laws should be query-able. Could laws and legislation be recorded in this precise format?
The purpose of this talk was to refine and extend the “databases are categories” idea from last time.

With schemas as categories, views, data migration, and integrating with PL become natural.

Recording world-views as ologs may yield an information network that is instantly compatible with database systems.

“It’s all categories and functors!” —
I hope people see category theory as a unified modeling language for information storage, processing, and transfer.

Thanks for hosting me and inviting me to speak!