Databases = Categories

David I. Spivak

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

Presented on 2010/09/16

My background

- Mathematics
- Thesis: algebraic topology
- Category theory and information science

The problem to be addressed

- Importance of databases:
  - Information is the currency of the 21st century.
  - The majority of computer cycles in the world operate on databases.
  - Organizing information is the only way to make sense of our world.
- Problems with databases:
  - They do not integrate well with programming languages.
  - Data migration is difficult and prone to errors.
  - Understanding a schema can be quite tedious.
- It's time to put databases on a firm mathematical footing.
  - Science always benefits from good mathematical underpinnings.
  - Need appropriate language and protocol for schemas and data.
  - But don't we already have that in the Relational Algebra?

The relational model

- Mathematical definition of relation:
    - Given sets (to be data types for columns) $A_1, A_2, \ldots, A_n$,
    - a relation on them is a subset $R \subset A_1 \times A_2 \times \cdots \times A_n$.
    - The elements of this subset are the rows of a table with the n specified columns.
- Relations are the current mathematical foundation of databases.
    - Brought to bear on databases by E.F. Codd.
    - Relations tell us what "tables" are,
    - they give us an algebra for taking subsets, unions, intersections, joins, etc., and
    - they give us a terminology and organizing principle.
- The mathematics of relations is over 150 years old.
    - The relational algebra is simply out of date.
    - It is not flexible enough to really describe what databases do.
    - Most importantly, relations are about single tables whereas databases are massive conglomerates of interconnected tables.

The status of database theory

- Juggling theory and practice
  - Databases have been successful since the beginning.
  - Because of a need for continuity, foundations have not been often reconsidered.
  - The basic theory (relations) is quite simple.
  - It has been extended again and again to give new organizational and functional capacities.
- Q: How well do the following fit in with the pure theory of relations?
  - Nulls
  - Skolem variables
  - Schemas
  - Queries
  - Views
  - Refactoring
  - Data migration
  - Optimizers

A need for unity

- Database management
  - Each of the above (nulls, Skolem, queries, views, etc.) is an essential aspect of how databases are used.
  - Each has its own mini-theory.
  - But the pieces don't fit together very well. Clunky.
- Programming languages
  - PL is more theoretical and unified.
  - Perhaps the reason that PL isn't integrating well with database is the lack of coherency on our side.
- Headaches.
  - The integration of PL and databases,
  - the integration of different aspects of a DBMS, and
  - the integration of different database systems —
  - all these headaches can be relieved by unifying database foundations.

Category theory: a unified modeling language

- A need for unity in mathematics
  - In the first half of the 20th century, a similar problem faced mathematics.
  - Each subfield had its own jargon and way of doing business.
  - It was noticed that there were similarities in all of them.
  - Different subfields needed each other in order to advance.
- Solution: category theory.
  - Invented in the 1940s to connect topology and algebra.
  - Powerful, expressive, and scalable, yet axiomatically simple.
  - Since its inception it has completely changed the way math is done.

Category theory in practice

- In mathematics categories abound:
  - Sets,
  - Partially ordered sets,
  - Graphs and trees,
  - Monoids, groups, finite groups
  - Topological spaces, vector spaces.
- Also used extensively in PL and linguistics
  - A category specifies a language.
  - Yet it is formal, structured, and well-defined.
  - As such it serves as a nice foundation for semantics, both in Linguistics and PL.
- Functors relate categories.
  - This is perhaps most important.
  - A precise definition for translating notions between categories.

My goal

- My goal was to find a categorical description of databases.
- A good language for discussing all aspects of databases in a unified way.
- It turns out that this is not only possible, it's quite simple.

Schemas are categories; categories are schemas

- The reason it's simple is that categories and database schemas do the same thing.
    - A schema gives a language for modeling a situation;
        - Types
        - Attributes
    - This is precisely what a category does.
        - Objects
        - Morphisms.
- Schema = Category, Instance = functor.
- In this talk, I'll explain these ideas and some consequences.

Outline of the talk

- Define categories, give examples, and show relation to schemas.
- Define functors, give examples, and show relation to data.
- Discuss morphisms of schemas and the associated data migration functors.

Categories

- Idea: A category models objects of a certain sort and the relationships between them.

$$\begin{tikzcd}
\bullet A \arrow[r, "f"] & \bullet B \arrow[r, bend left, "g"] \arrow[r, bend right, "h"'] & \bullet C \\
\bullet D \arrow[loop left, "i"] \arrow[r, bend left, "j"] & \bullet E \arrow[l, bend left, "k"] &
\end{tikzcd}$$

- Think of it like a graph: the nodes are objects and the arrows are relationships.
- Some paths can be equated with others (example: $j.k = i^3$).

Data of a category $\mathcal{C}$

A *category* $\mathcal{C}$ consists of the following data:

1. A set $\mathbf{Ob}(\mathcal{C})$, called *the set of objects of* $\mathcal{C}$.
   Objects $x \in \mathbf{Ob}(\mathcal{C})$ may be written as $\bullet^x$ or simply as x.
2. For each $x, y \in \mathbf{Ob}(\mathcal{C})$ a set $\mathbf{Arr}_{\mathcal{C}}(x, y)$, called *the set of arrows in* $\mathcal{C}$ *from* x *to* y.

   An element of $\mathbf{Arr}_{\mathcal{C}}(x, y)$ may be written $f\colon x \to y$ or $\bullet^x \xrightarrow{f} \bullet^y$.
3. For each $x, y, z \in \mathbf{Ob}(\mathcal{C})$ a *composition law*

$$\mathbf{Arr}_{\mathcal{C}}(x, y) \times \mathbf{Arr}_{\mathcal{C}}(y, z) \to \mathbf{Arr}_{\mathcal{C}}(x, z).$$

   We write $f.g = h$ to denote that following arrow f then arrow g is the same as following arrow h:

$$\begin{tikzcd}
\bullet^x \arrow[r, "f"] \arrow[dr, "h"'] & \bullet^y \arrow[d, "g"] \\
& \bullet^z
\end{tikzcd}$$

Rules for a category $\mathcal{C}$

These data must satisfy the following requirements:

1. For every object $y \in \mathbf{Ob}(\mathcal{C})$ there is an "identity arrow"

$$\mathrm{id}_y \colon y \to y$$

in $\mathbf{Arr}_{\mathcal{C}}(y, y)$ such that for any $f \colon x \to y$, the equations $\mathrm{id}_x.f = f$ and $f.\mathrm{id}_y = f$ hold:

$$\begin{tikzcd} \bullet^x \arrow[d, "\mathrm{id}_x"'] \arrow[dr, "f"] & \\ \bullet^x \arrow[r, "f"'] & \bullet^y \end{tikzcd} \qquad \begin{tikzcd} \bullet^x \arrow[r, "f"] \arrow[dr, "f"'] & \bullet^y \arrow[d, "\mathrm{id}_y"] \\ & \bullet^y \end{tikzcd}$$

2. Composition is *associative*. That is, given

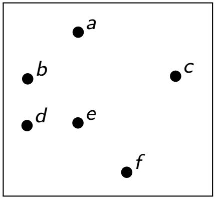
$$\bullet^w \xrightarrow{f} \bullet^x \xrightarrow{g} \bullet^y \xrightarrow{h} \bullet^z,$$

the following equation holds:

$$f.(g.h) = (f.g).h$$

Examples of categories 1: Sets

- Any set X can be considered as a category. Its objects are the elements of X and it has no arrows (except an identity arrow for each object).

$X =$ 

- **Set**. Objects are sets, arrows are total functions.

$$f : \boxed{\text{X}} \to \boxed{\text{Y}}$$

- **Set**$_*$. Objects are sets, arrows are partial functions.

Examples of categories 2: Posets and Graphs

- A partial order, $(X, \leq)$. Turn it into a category whose objects are the elements of X and $\mathbf{Arr}(x, y)$ has one element if $x \leq y$; else empty.
  - For example:

$$\begin{tikzcd} & B_1 \arrow[dr] & \\ A \arrow[ur] \arrow[rr] \arrow[dr] & & C \\ & B_2 \arrow[ur] & \end{tikzcd}$$

“$A \leq B_1, B_2 \leq C$”

  - Linear order:

$$[n] = \boxed{\bullet^0 \longrightarrow \bullet^1 \longrightarrow \cdots \longrightarrow \bullet^n}$$

- Any graph can be turned into a category in several ways.

$$\begin{tikzcd} \bullet^A \arrow[r, "f"] & \bullet^B \arrow[r, bend left, "g"] \arrow[r, bend right, "h"'] & \bullet^C \end{tikzcd}$$

Examples of categories 3: Others

- Functional programming languages
  - **Hask**. Objects are Haskell data-types, arrows are Haskell functions.
  - Similar for ML.
- The category of topological spaces.
- Linguistic categories. A tight connection with databases.

What is a database?

- A database consists of a bunch of tables and relationships between them.
- The rows of a table are called "records" or "tuples."
- The columns are called "attributes."
- A column may be "pure data" or may be a "key."
  - A table may have "foreign key columns" that link it to other tables.
  - A foreign key of table A links into the primary key of table B.
- A schema may have "business rules."

Foreign Keys

- Example:

Employee				
Employee_Id	**First**	**Last**	**Mgr**	**Dpt**
101	David	Hilbert	103	q10
102	Bertrand	Russell	102	x02
103	Alan	Turing	103	q10

Department		
Department_Id	**Name**	**Secr'y**
q10	Sales	101
x02	Production	102

- Note the primary key columns and foreign key columns.
- Perhaps we should enforce certain integrity constraints (business rules):
  - The manager of an employee E must be in the same department as E,
  - The secretary of a department D must be in D.

Data columns as foreign keys

- Theoretically we can consider a data-type as a 1-column table.
- Example:

String
a
b
⋮
z
aa
ab
⋮

- So any data column can be considered a foreign key to a 1-column table.
- Conclusion: each column in a table is a key – one primary, the rest foreign.

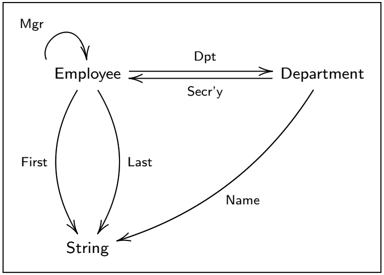
Categorical normal form

1. Every table T has a unique primary key column id_T, chosen at the outset. (Can be “row-number” or can be typed.)
2. Every data type D is considered as a 1-column table. The cells in id_D are the values of that data type.
3. Every column c of every table T refers to some other table T'. All values in the c-column can be found in the primary key column of T'.

Example again

Employee				
Employee_Id	**First**	**Last**	**Mgr**	**Dpt**
101	David	Hilbert	103	q10
102	Bertrand	Russell	102	x02
103	Alan	Turing	103	q10

Department		
Department_Id	**Name**	**Secr'y**
q10	Sales	101
x02	Production	102



$$\begin{tikzcd}
\text{Employee} \arrow[loop above, "\text{Mgr}"] \arrow[rr, shift left, "\text{Dpt}"] \arrow[d, bend right, "\text{First}"'] \arrow[d, bend left, "\text{Last}"] & & \text{Department} \arrow[ll, shift left, "\text{Secr'y}"] \arrow[lld, bend left, "\text{Name}"] \\
\text{String} & &
\end{tikzcd}$$

Database schema as a category

- A database schema is a system of tables linked by foreign keys.
- This is just a category!

$$\mathcal{C} = \boxed{\begin{tikzcd} \bullet\text{Employee} \arrow[loop above, "\text{Mgr}"] \arrow[r, shift left, "\text{Dpt}"] \arrow[d, bend right, "\text{First}"'] \arrow[d, bend left, "\text{Last}"] & \bullet\text{Department} \arrow[l, shift left, "\text{Secr'y}"] \arrow[dl, bend left, "\text{Name}"] \\ \bullet\text{String} & \end{tikzcd}}$$

- Objects are tables, arrows are columns.
- Primary key column of a table is identity arrow of an object.
- Declaring integrity constraints (e.g. `Mgr.Dpt=Dpt`) is declaring composition law.

Functors

- Idea: A functor is a graph morphism that is required to respect the composition law.
- Definition: A functor $F\colon \mathcal{C} \to \mathcal{D}$ consists of
    - A function $\mathbf{Ob}(F)\colon \mathbf{Ob}(\mathcal{C}) \to \mathbf{Ob}(\mathcal{D})$ and
    - a function $\mathbf{Arr}(F)\colon \mathbf{Arr}(\mathcal{C}) \to \mathbf{Arr}(\mathcal{D})$,

  that respect
    - the source and target of every arrow,
    - the identity arrow of every node, and
    - the composition law.
- Note that the composition of functors is a functor.

Examples of functors

- How many functors $F\colon \mathcal{C} \to \mathcal{D}$?

$$\mathcal{C}:=\ \begin{tikzcd} a \arrow[r, "e"] & b \end{tikzcd} \quad \xrightarrow{F} \quad \mathcal{D}:=\ \begin{tikzcd} X \arrow[r, "f"] & Y \arrow[r, bend left, "g"] \arrow[r, bend right, "h"'] & Z \end{tikzcd}$$

- Vals: **Hask** $\to$ **Set**. The set of values for each data type.
- $$\begin{tikzcd} A \arrow[r, "f"] & B \end{tikzcd} \longrightarrow \begin{tikzcd} C \arrow[loop above, "g"] \end{tikzcd}$$
- **Cat** is the category whose objects are categories and morphisms are functors.

Schema=Category, Data=Functor

- Let

$$\mathcal{C} = \begin{tikzcd}
\bullet\text{Employee} \arrow[loop above, "\text{Mgr}"] \arrow[r, shift left, "\text{Dpt}"] \arrow[d, bend right, "\text{First}"'] \arrow[d, bend left, "\text{Last}"] & \bullet\text{Department} \arrow[l, shift left, "\text{Secr'y}"] \arrow[dl, bend left, "\text{Name}"] \\
\bullet\text{String} &
\end{tikzcd}$$

$$\texttt{Mgr.Dpt} = \texttt{Dpt};$$

$$\texttt{Secr}'\texttt{y.Dpt} = \texttt{id\_Department}$$

- A functor $\delta\colon \mathcal{C} \to \textbf{Set}$ consists of
  - A set for each object of $\mathcal{C}$ and
  - a function for each arrow of $\mathcal{C}$, such that
  - the declared equations hold
- In other words, δ fills the schema with data.

Data as a functor

$\mathcal{C} =$

$$\begin{tikzcd}
\bullet E \arrow[loop left, "m"] \arrow[rr, shift left, "d"] \arrow[d, bend right, "f"'] \arrow[d, bend left, "l"] & & \bullet D \arrow[ll, shift left, "s"] \arrow[dll, "n"] \\
\bullet S & &
\end{tikzcd}$$

`m.d = d;`

`s.d = id_D`

$\delta\colon \mathcal{C} \longrightarrow \textbf{Set}$

$$\begin{tikzcd}
\boxed{\text{101,102,103}} \arrow[loop above, "m"] \arrow[rr, shift left, "d"] \arrow[d, bend right, "f"'] \arrow[d, bend left, "l"] & & \boxed{\text{q10,x02}} \arrow[ll, shift left, "s"] \arrow[dll, "n"] \\
\boxed{\text{a,b,...,Alan,Alao,...}} & &
\end{tikzcd}$$

- A category $\mathcal{C}$ is a schema. An object $x \in \textbf{Ob}(\mathcal{C})$ is a table.
- A functor $\delta\colon \mathcal{C} \longrightarrow \textbf{Set}$ fills the tables with compatible data.
- For each table x, the set $\delta(x)$ is its set of rows.
- The composition law in $\mathcal{C}$ is enforced by δ as business rules.

$\mathcal{C}$–**Set**, the category of instances of $\mathcal{C}$.

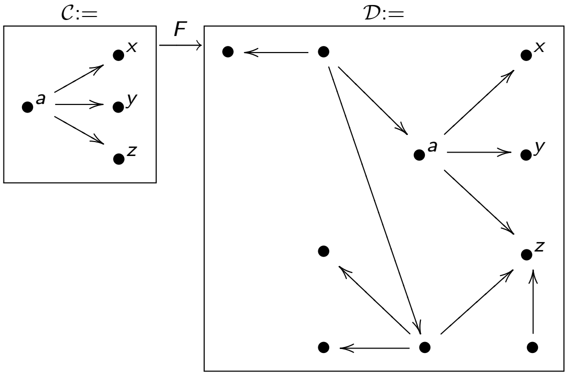
- Given a schema $\mathcal{C}$ there is a category of instances on $\mathcal{C}$; call it $\mathcal{C}$–**Set**.
- The objects of $\mathcal{C}$–**Set** are the instances of schema $\mathcal{C}$.
  - More precisely, an object of $\mathcal{C}$–**Set** is a functor $\delta\colon \mathcal{C} \to \mathbf{Set}$.
- The morphisms of $\mathcal{C}$–**Set** are updates.
  - Called *natural transformations of functors* $\delta \to \delta'$.
  - Not too hard, but not worth the effort here.

Recap

- Any category $\mathcal{C}$ is a database schema.
- Any functor $\delta\colon \mathcal{C} \to \mathbf{Set}$ is an instance of that schema.
- Updating instances takes place in the new category $\mathcal{C}$–**Set**.
- What does this do for you?

Morphisms between schemas

- A functor $F\colon \mathcal{C} \to \mathcal{D}$ is called *a morphism of schemas*.



- Many choices of F above, one of which is indicated.

A less obvious morphism

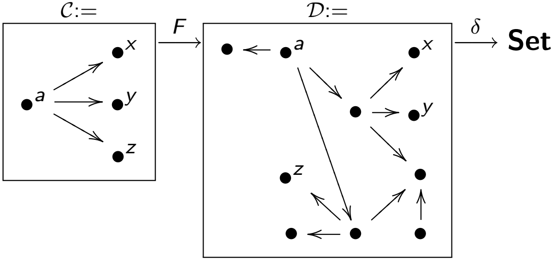
$$\mathcal{C}:= \quad \begin{tikzcd} & \bullet^{x} \\ \bullet^{a} \arrow[ur] \arrow[r] \arrow[dr] & \bullet^{y} \\ & \bullet^{z} \end{tikzcd} \quad \xrightarrow{F} \quad \mathcal{D}:= \quad \begin{tikzcd} \bullet & \bullet^{a} \arrow[l] \arrow[dr] \arrow[dddr] & & \bullet^{x} \\ & & \bullet \arrow[ur] \arrow[r] \arrow[dr] & \bullet^{y} \\ & \bullet^{z} & & \bullet \\ & \bullet & \bullet \arrow[l] \arrow[ul] \arrow[ur] & \bullet \arrow[u] \end{tikzcd}$$

Data migration 1: “pull-back”

- Given $F\colon \mathcal{C} \to \mathcal{D}$ and $\delta\colon \mathcal{D} \to \mathbf{Set}$,

$$\begin{tikzcd} \mathcal{C} \arrow[r, "F"] \arrow[rr, bend right, "F.\delta"'] & \mathcal{D} \arrow[r, "\delta"] & \mathbf{Set}, \end{tikzcd}$$

  compose to get $F.\delta\colon \mathcal{C} \to \mathbf{Set}$.
- Let δ vary (i.e. let updates take place).
- Now $F\colon \mathcal{C} \to \mathcal{D}$ gives a functor $\mathcal{D}$–$\mathbf{Set} \to \mathcal{C}$–$\mathbf{Set}$.



The existential and universal push-forwards

- Given a functor $F\colon \mathcal{C} \to \mathcal{D}$ we get a migration functor called

$$F^*\colon \mathcal{D}\text{–}\mathbf{Set} \to \mathcal{C}\text{–}\mathbf{Set}.$$

- This migration functor has a left adjoint $\exists_F$ and a right adjoint $\forall_F$.
- Both of these push data forward from $\mathcal{C}$ to $\mathcal{D}$

$$\exists_F\colon \mathcal{C}\text{–}\mathbf{Set} \to \mathcal{D}\text{–}\mathbf{Set} \qquad \forall_F\colon \mathcal{C}\text{–}\mathbf{Set} \to \mathcal{D}\text{–}\mathbf{Set}.$$

- I won't go into how these work but just give some quick examples.

The universal push-forward $\forall_F$ makes joins

$$\mathcal{C}:= \begin{tikzcd} & a_1 \arrow[dl] \arrow[d] \arrow[dr] \arrow[rr] & & a_2 \arrow[d] \arrow[r] & a_3 \arrow[d] \arrow[dr] & \\ b_1 & b_2 & b_3 & c_1 & d_1 & d_2 \end{tikzcd} \xrightarrow{F} \mathcal{D}:= \begin{tikzcd} & a \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] \arrow[drrr] \arrow[drrrr] & & & & \\ b_1 & b_2 & b_3 & c_1 & d_1 & d_2 \end{tikzcd}$$

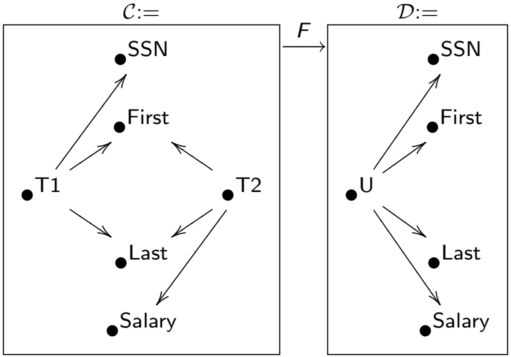
- We begin with three tables (with 3,1, and 2 data columns respectively) arranged in a detail to master hierarchy.
- Apply the universal push-forward $\forall_F$ to join them into one table.

More joins using $\forall_F$

$$\begin{tikzcd}
 & \bullet \text{SSN} & \\
 & \bullet \text{First} & \\
\bullet \text{T1} \arrow[uur] \arrow[ur] \arrow[dr] & & \bullet \text{T2} \arrow[ul] \arrow[dl] \arrow[ddl] \\
 & \bullet \text{Last} & \\
 & \bullet \text{Salary} &
\end{tikzcd}
\xrightarrow{F}
\begin{tikzcd}
 & \bullet \text{SSN} \\
 & \bullet \text{First} \\
\bullet \text{U} \arrow[uur] \arrow[ur] \arrow[dr] \arrow[ddr] & \\
 & \bullet \text{Last} \\
 & \bullet \text{Salary}
\end{tikzcd}$$

$$\begin{tikzcd}
\bullet \text{Employee} \arrow[d] & \bullet \text{Converter} \arrow[dl] \arrow[dr] & \\
\bullet \text{USD} & & \bullet \text{Euro}
\end{tikzcd}
\xrightarrow{F}
\begin{tikzcd}
\bullet \text{Employee} \arrow[d] \\
\bullet \text{Euro}
\end{tikzcd}$$

The existential push-forward $\exists_F$ makes unions



- Given any instance $\delta\colon \mathcal{C} \to \textbf{Set}$, get an instance $\exists_F \delta\colon \mathcal{D} \to \textbf{Set}$.
- The rows in table $\bullet^{\text{U}}$ will be the union of the rows in $\bullet^{\text{T1}}$ and $\bullet^{\text{T2}}$.
- It will automatically use Skolem variables for the unknown cells.

Advantage from mathematical basis

- Unity
  - Putting large swath of database concepts under one framework.
  - This framework works nicely with mathematics, programming languages, lingusitics.
  - Opens up a world of possibility.
- Visualizability
  - The ability to visualize SQL statements in terms of graph (category) morphisms.
  - Should be easy to create a system to convert drawings of categories into schemas.
  - This allows a larger set of “every day” people to work with database systems.

Advantage of rigor / “soundness.”

- The ability to encode business rules in the structure –
  - This ensures a higher quality in data migration.
  - It also makes for more seamless transitions to new DBAs.
- Rules for schema migration are more precise.
  - Typically there are many rules for database mapping.
  - These rules are static and over-cautious.
  - With good foundation, one can decide more dynamically what is ok.
- One can employ theorem provers and checkers.
  - The schema and the data have a mathematical structure.
  - Theorem checkers can support your results or stated characteristics.
  - Prove that a certain query-plan will be fastest in a local situation.

Summary

- Category theory is useful.
    - It can be used to model any well-defined situation.
    - It unifies and clarifies.
    - It is about as hard to learn as discrete mathematics is.
- Categories and database schemas are the same thing!
    - No wonder categories are useful in modeling.
- Mathematical language has many advantages.
    - It is generally worth the effort to learn.
- I am available to work on this connection.