Categorical databases

David I. Spivak

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

Presented on 2014/02/28
at Oracle
We live in a world of interacting entities.

- Humans interact with machines.
- Companies interact with companies.
- Databases interact with applications.
- And so on, ad infinitum.

Each entity has its own internal language optimized for operating in some context.

Interaction between entities is mediated by an interaction between their languages.

Communication is the successful transfer of information through interaction.

We need to find a formal underpinning for meaningful communication.
Databases as languages

- A database schema serves as a language, optimized for the operation of some entity in some context.
- But today, entities and contexts change more and more frequently.
- We need to transfer information not only to others, but to our later, different selves.
- We should think of data management as one simple idea: a translation of information from one form to another.
  - An ETL process translates from normalized form to warehouse form.
  - A data migration between databases translates from one schematic form to another.
  - A query against a database translates from all-purpose form to specific-purpose form.
  - An update does not change the schematic form, but still procedurally translates information.
There is an fundamental connection between databases and categories.

I propose that:

- Category theory can simplify how we think about and use databases.
- We can clearly see all the working parts and how they fit together.
- Powerful theorems can be brought to bear on classical DB problems.
Relational databases are reliable, scalable, and popular.
They are provably reliable to the extent that they strictly adhere to the underlying mathematics.
Make a distinction between
- the system you know and love, vs.
- the relational model, as a mathematical foundation for this system.
You’re not really using the relational model.

- Current implementations have departed from the strict relational formalism:
  - Tables may not be relational (duplicates, e.g. from a query).
  - Nulls (and labeled nulls) are commonly used.
- The theory of relations (150 years old) is not adequate to mathematically describe modern DBMS.
- The relational model is either clumsy or completely absent in describing:
  - Foreign keys,
  - Updates,
  - Schema mappings and data migration,
  - Distributed databases, etc.
- Databases have been intuitively moving toward what’s best described with a more modern mathematical foundation.
Category theory gives better description

- Category theory (CT) does a better job of describing what’s already being done in DBMS.
  - Puts functional dependencies and foreign keys front and center.
  - Allows non-relational tables (e.g. duplicates in a query).
  - Labeled nulls and semi-structured data fit in neatly.
- CT offers guidance for all sorts of information hand-off:
  - Foreign key as a handoff from one table to another.
  - Query as a handoff from general schema to specific schema.
  - ETL as a handoff from normalized schema to unnormalized schema.
  - Data migration as information handoff from one schema to another.
  - Update as information handoff from a schema to itself.
  - Applications as information handoff between programming language and database.
What is category theory?

- Category theory is the mathematics of information handoff.
  - That is, CT is about integrity of relationship under change of form.
- Since its invention in the early 1940s, category theory has revolutionized math.
- It’s like set theory and logic, except less floppy, more principles-based.
- Category theory has been proposed as a new foundation for mathematics (to replace set theory).
- It was invented to build bridges between disparate branches of math by distilling the essence of mathematical structure.
Branching out

- Category theory naturally fosters connections between disparate fields.
- It has branched out of math and into physics, linguistics, and materials science.
- It has had much success in the theory of programming languages.
- The pure category-theoretic concept of monads has vastly extended the reach of functional programming.
- Can category theory improve how we think about databases?
Schemas are categories, categories are schemas

- The connection between databases and categories is simple and strong.
- **Reason**: categories and database schemas do the same thing.
  - A schema gives a framework for modeling a situation;
    - Tables
    - Attributes
  - This is precisely what a category does.
    - Objects
    - Arrows.
  - They both model how entities within a given context interact.
- **The functorial data model is what you get when you demand:**

  \[ \text{Schema} = \text{Category}. \]
The category of categories

- A database schema can be modeled as a finite category.
  - Both model the interactions between tables (via columns).
  - The semantics of this category is captured by instances.
  - Each instance describes the relationship between rows in different tables.
- But there is also a *category of categories*.
  - This models the interactions between database schemas.
  - The semantics of this category is understood as data migration, querying etc.
  - Data migration describes the relationship between instances on different schemas.
- Instead of describing all this today, let’s just work on the basic idea.
  - My colleague believed it would be better to demo the tool than get into the math.
  - I’ll be available for questions later, and there are papers online.
The basic idea

- In the **functorial data model**, a database schema is a special kind of entity-relationship (ER) diagram.

\[
\text{Emp}.	ext{manager}.	ext{worksIn} = \text{Emp}.	ext{worksIn} \quad \text{Dept}.	ext{secretary}.	ext{worksIn} = \text{Dept}
\]

<table>
<thead>
<tr>
<th>Emp</th>
<th>mgr</th>
<th>works</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>103</td>
<td>q10</td>
<td>Al</td>
<td>Akin</td>
</tr>
<tr>
<td>102</td>
<td>102</td>
<td>x02</td>
<td>Bob</td>
<td>Bo</td>
</tr>
<tr>
<td>103</td>
<td>103</td>
<td>q10</td>
<td>Carl</td>
<td>Cork</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dept</th>
<th>sec</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>q10</td>
<td>102</td>
<td>CS</td>
</tr>
<tr>
<td>x02</td>
<td>101</td>
<td>Math</td>
</tr>
</tbody>
</table>
The basic idea, continued

- Each node represents an entity set.
- Each entity is identified by a globally unique ID and has some attributes (strings, integers, etc).
- Each directed edge represents a foreign key.
- Data integrity constraints are path equalities.
- Instances are **always and only** considered up to isomorphism of IDs and equality of attributes.
- No nulls.
- We usually think of each attribute and edge as a binary table.
The functorial data model can be studied with the regular tools of relational database theory.

The functorial data model can also be studied with category theory.
- Category theory reveals many additional useful properties of functorial schemas and instances that are invisible to traditional database theory.

The purpose of this talk is to demonstrate these properties without getting into the mathematics of category theory.

To do this, we will describe FQL, a functorial query language, and the relationship between FQL and SQL.

Everything in this talk has been implemented by Ryan Wisnesky.
- Download the FQL IDE from:
  http://categoricaldata.net/fql.html
A schema mapping $F : S \rightarrow T$ is a constraint-respecting mapping:

$$\text{nodes}(S) \rightarrow \text{nodes}(T) \quad \text{edges}(S) \rightarrow \text{paths}(T)$$

and it induces three data migration operations:

- $\Delta_F : T \rightarrow \text{inst} \rightarrow S \rightarrow \text{inst}$ (like projection)
- $\Sigma_F : S \rightarrow \text{inst} \rightarrow T \rightarrow \text{inst}$ (like union)
- $\Pi_F : S \rightarrow \text{inst} \rightarrow T \rightarrow \text{inst}$ (like join)
I’ll give some examples and then demo them in the FQL IDE.

In each case, I’ll show a couple FQL schemas $S, T$ and an FQL mapping $F : S \to T$ between them.

- I’ll talk about what the three data migration functors, $\Delta_F, \Sigma_F, \Pi_F$ do in each case.
- I’ll show some EDs (embedded dependencies) that would have the same result under the chase.
Example 1

\[ S := \begin{array}{c}
\bullet \\
\circ \rightarrow \bullet \\
\circ \rightarrow \bullet \\
\end{array} \]

\[ F \Rightarrow \begin{array}{c}
\bullet \\
\circ \rightarrow \bullet \\
\circ \rightarrow \bullet \\
\end{array} := T \]

- \( \Delta_F : T-\text{Inst} \rightarrow S-\text{Inst} \) copies \( n \) into \( n_1 \) and \( n_2 \), and \( a \) into \( a_1 \) and \( a_2 \).
  \[
  a(x, y) \rightarrow a_1(x, y) \land a_2(x, y)
  \]

- \( \Pi_F : S-\text{Inst} \rightarrow T-\text{Inst} \) joins \( a_1 \) and \( a_2 \) into \( a \), creating a fresh ID for each tuple.
  \[
  a_1(x, y) \land a_2(x', y) \rightarrow \exists z, a(z, y)
  \]

- \( \Sigma_F : S-\text{Inst} \rightarrow T-\text{Inst} \) unions \( a_1 \) and \( a_2 \) into \( a \).
  \[
  a_1(x, y) \rightarrow a(x, y) \quad a_2(x, y) \rightarrow a(x, y)
  \]
Demo Example 1 in FQL IDE

\[
\text{schema } S = \{ \text{nodes } n_1, n_2; \text{ attributes } a_1 : n_1 \rightarrow \text{ string}, a_2 : n_2 \rightarrow \text{ string}; \\
\text{ arrows; equations; } \} \\
\]

\[
\text{schema } T = \{ \text{nodes } n; \text{ attributes } a : n \rightarrow \text{ string}; \text{ arrows; equations;} \}
\]

\[
\text{mapping } F = \{ \\
\text{ nodes } n_1 \rightarrow n, n_2 \rightarrow n; \\
\text{ attributes } a_1 \rightarrow a, a_2 \rightarrow a; \\
\text{ arrows; } \} : S \rightarrow T
\]

\[
\text{instance } I = \{ \\
\text{ nodes } n_1 \rightarrow \{0,1,2\}, n_2 \rightarrow \{3,4\}; \\
\text{ attributes } a_1 \rightarrow \{(0,\text{alpha}),(1,\text{beta}),(2,\text{gamma})\}, \\
\text{ a2 } \rightarrow \{(3,\text{alpha}),(4,\text{upsilon})\}; \\
\text{ arrows; } \} : S
\]

\[
\text{instance } \pi_F_I = \pi F I \\
\text{instance } \sigma_F_I = \sigma F I
\]
Example 2 (adds an edge to Example 1)

\[
S := \begin{array}{c}
\circ & \rightarrow & n_1 \\
\bullet & \downarrow & \circ \\
\circ & \rightarrow & n_2 \\
\end{array}
\]

\[
F \rightarrow \begin{array}{c}
\circ & \rightarrow & n \\
\bullet & \downarrow & \circ \\
\end{array} := T
\]

- \( \Delta_F : T - \text{Inst} \rightarrow S - \text{Inst} \) puts the identity function into \( f \):
  \[
a(x, y) \rightarrow a_1(x, y) \land a_2(x, y) \land f(x, x)
\]

- \( \Pi_F : S - \text{Inst} \rightarrow T - \text{Inst} \) will join \( a_1, a_2, \) and \( f \):
  \[
a_1(x, y) \land a_2(x', y) \land f(x, x') \rightarrow \exists z, a(z, y)
\]

- \( \Sigma_F : S - \text{Inst} \rightarrow T - \text{Inst} \) will union \( a_1 \) and \( a_2 \) into \( a \), requiring each ID \( x \) in \( n_1 \) to have the same \( a \) attribute as \( f(x) \) in \( n_2 \).
  \[
a_1(x, y) \rightarrow a(x, y) \quad a_2(x, y) \rightarrow a(x, y) \quad f(x, y) \rightarrow \exists z, a(x, z) \land a(y, z)
\]

Consequently, it is possible for \( \Sigma_F \) to fail if it must equate two distinct constants (like “alice” and “bob”).
schema $S = \{ \text{nodes } n_1, n_2; \text{ attributes } a_1 : n_1 \rightarrow \text{string}, a_2 : n_2 \rightarrow \text{string};$

arrows $f : n_1 \rightarrow n_2; \text{ equations}; \}$

schema $T = \{ \text{nodes } n; \text{ attributes } a : n \rightarrow \text{string}; \text{ arrows; equations}; \}$

mapping $F = \{$

nodes $n_1 \rightarrow n, n_2 \rightarrow n;$

attributes $a_1 \rightarrow a, a_2 \rightarrow a;$

arrows $f \rightarrow n; \} : S \rightarrow T$

instance $I = \{$

nodes $n_1 \rightarrow \{0,1,2\}, n_2 \rightarrow \{3,4\};$

attributes $a_1 \rightarrow \{(0,\text{alpha}),(1,\text{alpha}),(2,\text{alpha})\},$

$a_2 \rightarrow \{(3,\text{alpha}),(4,\text{upsilon})\};$

arrows $f \rightarrow \{(0,3),(1,3),(2,3)\}; \} : S$

instance $\pi_{F,I} = \pi F I$
instance $\sigma_{F,I} = \sigma F I$
Example 3

\[ S := \begin{array}{c}
\text{Emp} \\
\bullet
\end{array} \xrightarrow{F} \begin{array}{c}
\text{Mgr} \\
\bullet
\end{array} := T \]

- \( \Pi_F \) will migrate into \( \text{Mgr} \) only those \( \text{Emps} \) that are their own \( \text{mgr} \).
- \( \Sigma_F \) will migrate into \( \text{Mgr} \) the “management groups” of \( \text{Emp} \), i.e. equivalence classes of \( \text{Emps} \) modulo the equivalence relation generated by \( \text{mgr} \).
- Key point of the examples: Functorial data migration operators are very expressive.
  - Note that none of these examples used path equality constraints.
  - We can be even more expressive if we employ them.
Functorial data migrations have a useful normal form:

$$\Sigma_F \circ \Pi_{F'} \circ \Delta_{F''}$$

Caveat: $F$ must obey a restriction that (roughly) it only takes unions of tables that are “union compatible.”

We call data migrations above the above form **FQL queries**.

Analogously, unions of conjunctive queries are a useful normal form for relational algebra.
Key results:

FQL queries can all be written in the following form:

$$\Sigma_F \circ \Pi_F' \circ \Delta_{F''}$$

- FQL queries are closed under composition.
  - Meaning we can implement compositions without materializing intermediate results.
- Unions of conjunctive queries can be implemented in FQL.
  - Natively it has bag semantics, which can be useful.
  - One can also obtain set semantics, after some simple post-processing.
- Every FQL query can be implemented as a union of conjunctive queries under set semantics
  - Note that we need an operation for creating globally unique IDs (e.g., using SQL auto-generated row IDs).
As you can see, the FQL IDE generates SQL, displayed at the bottom.

- Note that the FQL IDE is executing that SQL via JDBC on a 3rd-party SQL engine.

We can also see an operation category theory produces for free, the category of elements.

- This operation is interesting because it converts any instance to a schema.
- Time permitting, I’ll discuss this at the end of the talk.
Let's look at how to translate unions of conjunctive queries (SPCU) to FQL.

- The SELECT and FROM clauses are $\Delta$, which gathers the required tables.
- the WHERE clause is $\Pi$, which joins.
- the UNION clause (bag semantics) is $\Sigma$, which takes unions.
  (For set semantics, we post-process with something called RELATIONALIZE.)

Show the active domain, and remark that it is expensive to compute.

We can also translate SQL schemas to FQL

- Each table must have a single primary key column.
- Any number of foreign key constraints.
FQL vs SQL

Since FQL compiles to SQL, why not just write in SQL?

- FQL query results are entire databases, complete with foreign keys.
  - SQL only computes one table at a time.
- With FQL, output databases are guaranteed to obey their path-equality constraints.
  - One can write SQL queries that do not satisfy the necessary constraints.
  - FQL uses the foreign-key architecture of the target schema as part of its query semantics.

Since SQL compiles to FQL, why not just write in FQL?

- Currently, schemas and instances imported from SQL are not appropriately native to FQL.
  - We saw that we can compute unions of conjunctive queries, so it’s not horrible.
  - But to get the correct behavior, they are encoded using an expensive “active domain” construction that is not feasible in practice.
When the $F$ in $\Sigma_F$ is not a “discrete op-fibration”, $\Sigma_F$ cannot be computed by SQL.

Key result: it can be computed by chasing a set of embedded dependencies.

The semantics of such $\Sigma_F$ is similar to that of Clio or other data exchange systems, but

- $\Sigma_F$ has better properties (e.g., closure under composition)
- $\Sigma_F$ is more powerful (e.g., can compute connected components of a graph)
I will demo an example in which we union along foreign keys.

- Amphibians has a foreign key to land animals and to water animals.
- We add a new table (animals) and a new path equation.
- FQL generates some EDs.
- We get the right number of animals (7).
- Clio computes 9 animals: it ignores the path equality constraint, because it only handles TGDs.

Amphibian.isL.Lis = Amphibian.isW.Wis
How’s the time?

Shall we skip to the summary now, or keep going with RDF?
Category theory has an operation for converting any instance into a schema.

- Since all schemas are graphs, what graph do you get?
- Answer: the RDF graph.

We have FQL emitting OWL and RDF.

- Functorial schemas (without path equations) can be output as OWL.
- Every functorial instance is naturally encoded as RDF.
- And of course, the RDF is verified against its OWL schema.

Demo: Employees example

- Note the RDF and OWL output.
- Note the category of elements.
Conclusions:

- By restricting to functorial schemas and instances, we gain many useful properties:
  - Schemas
    - are ER diagrams
    - have data integrity constraints built-in
  - Data migrations
    - are weak inverses to each other
    - operate on entire databases
    - preserve constraints
    - can implement unions of conjunctive queries
    - are closed under composition (exception: “special Σ”)
    - are implementable in SQL (exception: “special Σ”)
  - All these properties were discovered through category theory.
Current work:
- The view-update problem for FQL.
- Additional programming constructs for FQL (e.g., exponentials).

Future work:
- Integrating FQL with a general-purpose programming language.
- A “native”, non-SQL implementation of FQL.
- Grouping, aggregation, nesting, difference, nulls.
- Optimization.
Thanks for inviting me to speak!
Programming in FQL

- The functorial data model admits many useful operations on schemas and mappings: products, co-products, etc.
- In fact, there is enough structure to interpret the simply-typed λ-calculus (STLC).
  - Key result: every type in the STLC denotes a schema, and every (open) term denotes a mapping.
- For each schema $S$, the functorial data model admits many useful operations on $S$-instances and $S$-homomorphisms: products, co-products, etc.
- In fact, there is enough structure to interpret higher-order logic (HOL).
  - Key result: every type in HOL denotes an $S$-instance, and every (open) term denotes an $S$-homomorphism.
- Show FQL products and co-products example.
We can always encode arbitrary relational databases as functorial instances using an explicit active domain construction. Consider a relational schema with two relations $R(c_1, \ldots, c_n)$ and $R'(c'_1, \ldots, c'_n)$.
Let $R$ be a table. We can express $\pi_{i_1, \ldots, i_k} R$ using $\Delta_F$

This construction is only appropriate for bag semantics because $\pi R$ will have the same number of rows as $R$. 
Let $R$ be a table. We can express $\sigma_{i=j} R$ using $\Pi_F$:

Here $F(c_i) = F(c_j) = s$. 

$$
\begin{align*}
\begin{array}{c}
\text{c}_1 & \cdots & \text{c}_i & \cdots & \text{c}_j & \cdots & \text{c}_n \\
\end{array}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c}
\text{R} \\
\text{D} \\
\end{array}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c}
\text{c}_1 & \cdots & \text{c}_i & \cdots & \text{c}_j & \cdots & \text{c}_n \\
\end{array}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c}
\text{R} \\
\text{D} \\
\end{array}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c}
\text{s} \\
\text{c}_1 & \cdots & \text{c}_n \\
\end{array}
\end{align*}
$$

$$
\begin{align*}
\begin{array}{c}
\text{s} \\
\text{D} \\
\end{array}
\end{align*}
$$
Let $R$ and $R'$ be tables. We can express $R \times R'$ as $\Pi_F$

$$
\begin{array}{c}
\begin{array}{c}
R \\
\bullet \\
\vdots \\
\circ \\
D \\
\circ \\
\bullet \\
R'
\end{array} & \begin{array}{c}
\begin{array}{c}
\rightarrow \\
c_n \\
\cdots \\
\rightarrow \\
c_{n'} \\
\downarrow \\
\leftarrow \\
\bullet \\
D \\
\circ \\
\leftarrow \\
\bullet \\
R'
\end{array}
\end{array} \\
\begin{array}{c}
\begin{array}{c}
\leftarrow \\
c_1 \\
\downarrow \\
\rightarrow \\
\circ \\
D \\
\circ \\
\bullet \\
R'
\end{array}
\end{array}
\end{array}
$$

$$
\begin{array}{c}
\begin{array}{c}
R \times R' \\
\bullet \\
\vdots \\
\circ \\
D \\
\circ \\
\bullet \\
R'
\end{array} & \begin{array}{c}
\begin{array}{c}
\begin{array}{c}
\leftarrow \\
c_1 \\
\cdots \\
\leftarrow \\
c_n \\
\cdots \\
\leftarrow \\
c_{n'} \\
\circ \\
D \\
\circ \\
\bullet \\
R'
\end{array}
\end{array} \\
\begin{array}{c}
\begin{array}{c}
\leftarrow \\
c_1 \\
\cdots \\
\leftarrow \\
c_n \\
\cdots \\
\leftarrow \\
c_{n'} \\
\circ \\
D \\
\circ \\
\bullet \\
R'
\end{array}
\end{array}
\end{array}
\end{array}
$$
FQL as SQL – implementing the data migration functors

- $\Delta$ can be implemented with conjunctive queries and ID-generation.
  - Target node tables are copied from the source. Target edge/attribute tables are populated by compositions of source edge/attribute tables. ID-generation only used to restore globally unique ID requirement.

- $\Sigma$ can be implemented with unions of conjunctive queries and ID-generation.
  - Algorithm is similar to a “union of $\Delta$s”.

- $\Pi$ can be implemented with conjunctive queries and ID-generation.
  - The most difficult to implement. Requires computing large “limit” tables that are similar to “join all”. ID-generation used to create IDs for the rows in the limit tables.