Categorical databases

David I. Spivak

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

Presented on 2014/02/28
at Oracle

Purpose of the talk

There is an fundamental connection between databases and categories.

- 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.

The pros and cons of relational databases

- 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 does not offer guidance for schema mappings and data migration.
- 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.
- All columns of a table are the same type of thing. It's simpler.
- CT offers guidance for schema mapping and data migration.
- It offers the opportunity to deeply integrate programming and data.
- Theorems within category theory, and links to other branches of math (e.g. topology), can be used in databases.

What is category theory?

- 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?

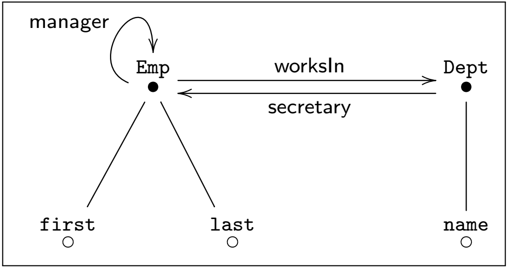
Schemas are categories, categories are schemas

- 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 basic idea

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

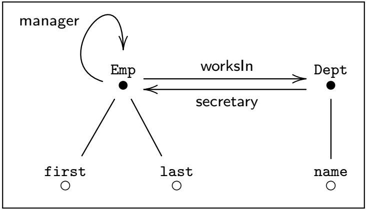


$$\mathtt{Emp.manager.worksIn} = \mathtt{Emp.worksIn} \qquad \mathtt{Dept.secretary.worksIn} = \mathtt{Dept}$$

Emp				
Emp	**mgr**	**works**	**first**	**last**
101	103	q10	Al	Akin
102	102	x02	Bob	Bo
103	103	q10	Carl	Cork

Dept		
Dept	**sec**	**name**
q10	102	CS
x02	101	Math

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.

Outline

- 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.
- Everything in this talk has been implemented by Ryan Wisnesky.
  - Download the FQL IDE from: `http://wisnesky.net/fql.html`

Schema mappings and associated operations

- A **schema mapping** $F : S \to T$ is a constraint-respecting mapping:

$$nodes(S) \to nodes(T) \qquad edges(S) \to paths(T)$$

  and it induces three **data migration** operations:
  - $\Delta_F : T - inst \to S - inst$ (like projection)
  - $\Sigma_F : S - inst \to T - inst$ (like union)
  - $\Pi_F : S - inst \to T - inst$ (like join)

Demos

- I'll give some examples and then demo them in the FQL IDE.
- In each case, I'll show a couple FQL schemas $\mathbf{S}, \mathbf{T}$ and an FQL mapping $F\colon \mathbf{S} \to \mathbf{T}$ 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

$$S := \begin{array}{c} \overset{\texttt{a1}}{\circ} \text{ --- } \overset{\texttt{n1}}{\bullet} \\ \\ \overset{\texttt{a2}}{\circ} \text{ --- } \overset{\texttt{n2}}{\bullet} \end{array} \qquad \xrightarrow{F} \qquad \overset{\texttt{a}}{\circ} \text{ --- } \overset{\texttt{n}}{\bullet} := T$$

- $\Delta_F \colon T\text{–}\mathbf{Inst} \to S\text{–}\mathbf{Inst}$ copies `n` into `n1` and `n2`, and `a` into `a1` and `a2`.

$$\mathtt{a}(x, y) \to \mathtt{a1}(x, y) \wedge \mathtt{a2}(x, y)$$

- $\Pi_F \colon S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$ joins `a1` and `a2` into `a`, creating a fresh ID for each tuple.

$$\mathtt{a1}(x, y) \wedge \mathtt{a2}(x', y) \to \exists z, \mathtt{a}(z, y)$$

- $\Sigma_F \colon S\text{–}\mathbf{Inst} \to T\text{–}\mathbf{Inst}$ unions `a1` and `a2` into `a`.

$$\mathtt{a1}(x, y) \to \mathtt{a}(x, y) \qquad \mathtt{a2}(x, y) \to \mathtt{a}(x, y)$$

Demo Example 1 in FQL IDE

```
schema S = {nodes n1, n2; attributes a1 : n1 -> string, a2 : n2 -> string;
  arrows; equations; }

schema T = {nodes n; attributes a : n -> string; arrows; equations;}

mapping F = {
 nodes n1 -> n, n2 -> n;
 attributes a1 -> a, a2 -> a;
 arrows; } : S -> T

instance I = {
 nodes n1 -> {0,1,2}, n2 -> {3,4};
 attributes a1 -> {(0,alpha),(1,beta),(2,gamma)},
   a2 -> {(3,alpha),(4,upsilon)};
 arrows; } : S

instance pi_F_I = pi F I
instance sigma_F_I = sigma F I
```

Example 2 (adds an edge to Example 1)

$$S := \begin{tikzcd} \overset{\texttt{a1}}{\circ} \arrow[r, dash] & \overset{\texttt{n1}}{\bullet} \arrow[d, "f"] \\ \overset{\texttt{a2}}{\circ} \arrow[r, dash] & \overset{\texttt{n2}}{\bullet} \end{tikzcd} \qquad \xrightarrow{F} \qquad \begin{tikzcd} \overset{\texttt{a}}{\circ} \arrow[r, dash] & \overset{\texttt{n}}{\bullet} \end{tikzcd} := T$$

- $\Delta_F : T$–**Inst** $\to S$–**Inst** puts the identity function into `f`:

$$\texttt{a}(x, y) \to \texttt{a1}(x, y) \wedge \texttt{a2}(x, y) \wedge \texttt{f}(x, x)$$

- $\Pi_F : S$–**Inst** $\to T$–**Inst** will join `a1`, `a2`, and `f`:

$$\texttt{a1}(x, y) \wedge \texttt{a2}(x', y) \wedge \texttt{f}(x, x') \to \exists z, \texttt{a}(z, y)$$

- $\Sigma_F : S$–**Inst** $\to T$–**Inst** will union `a1` and `a2` into `a`, requiring each ID x in `n1` to have the same `a` attribute as $\texttt{f}(x)$ in `n2`.

  $$\texttt{a1}(x, y) \to \texttt{a}(x, y) \quad \texttt{a2}(x, y) \to \texttt{a}(x, y) \quad \texttt{f}(x, y) \to \exists z, \texttt{a}(x, z) \wedge \texttt{a}(y, z)$$

  Consequently, it is possible for Σ_F to fail if it must equate two distinct constants (like "alice" and "bob").

Demo Example 2 in FQL IDE

```
schema S = {nodes n1, n2; attributes a1 : n1 -> string, a2 : n2 -> string;
  arrows f : n1 -> n2; equations; }

schema T = {nodes n; attributes a : n -> string; arrows; equations;}

mapping F = {
 nodes n1 -> n, n2 -> n;
 attributes a1 -> a, a2 -> a;
 arrows f -> n; } : S -> T

instance I = {
 nodes n1 -> {0,1,2}, n2 -> {3,4};
 attributes a1 -> {(0,alpha),(1,alpha),(2,alpha)},
   a2 -> {(3,alpha),(4,upsilon)};
 arrows f -> {(0,3),(1,3),(2,3)}; } : S

instance pi_F_I = pi F I
instance sigma_F_I = sigma F I
```

Example 3

$$S := \begin{tikzcd} \overset{\texttt{Emp}}{\bullet} \arrow[loop left, "\texttt{mgr}"] \end{tikzcd} \quad \xrightarrow{F} \quad \overset{\texttt{SelfMgr}}{\bullet} := T$$

- Π_F will migrate into `SelfMgr` only those `Emp`s that are their own `mgr`.
- Σ_F will migrate into `SelfMgr` the "management groups" of `Emp`, i.e. equivalence classes of `Emp`s modulo the equivalence relation generated by `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.

FQL - A Functorial Query Language

- 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).

Demo - People

- 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.

Demo - RA to FQL

- Let's look at how to translate unions of conjunctive queries (SPCU) to FQL.
  - The SELECT and FROM clauses are Δ, which gathers the required tables.
  - the WHERE clause is Π, which joins.
  - the UNION clause (bag semantics) is Σ, 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?

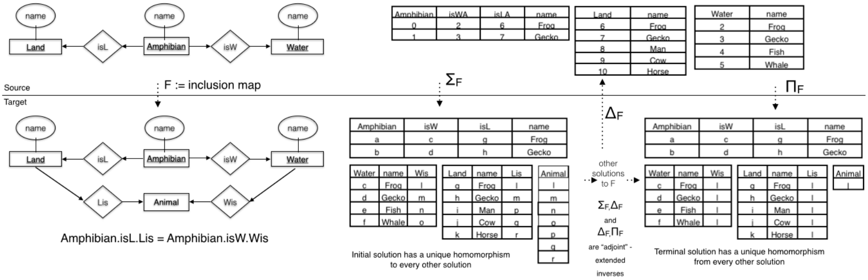
- Currently, schemas and instances imported from SQL are not appropriately native to FQL.
  - We saw that we can compute unions of conjunctive queries, so it's not horrible.
  - But to get the correct behavior, they are encoded using an expensive "active domain" construction that is not feasible in practice.

FQL and Data Exchange

- When the F in Σ_F is not a "discrete op-fibration", Σ_F cannot be computed by SQL.
- Key result: it can be computed by chasing a set of embedded dependencies.
- The semantics of such Σ_F is similar to that of Clio or other data exchange systems, but
  - Σ_F has better properties (e.g., closure under composition)
  - Σ_F is more powerful (e.g., can compute connected components of a graph)

Demo: Data Exchange

- 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.



How's the time?

Shall we skip to the summary now, or keep going with RDF?

FQL to 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.

A theorem of category theory

- Given an RDF triple store X, we can pull off all the labels and just get the graph.
- Recall that given an FQL schema and instance, you can produce an RDF graph.
- Question: what FQL schemas and instances produce graph X?
  - Is there a smallest schema with an instance that gives X?
- Theorem of category theory:
  - No, in general there is no smallest schema for X.

Interpretation and editorial

- So there's no smallest schema that produces a given RDF triple store.
- What does this mean?
  - RDF is unstructured in that it can arise from many different relational schemas.
  - There is no "best perspective" on RDF data.
- Editorial: "Don't move to schema-less, move to schema-more."
  - Coined by my colleague at Johnson & Johnson in response to NoSQL.
  - With categorical databases, it's easy to create schemas and functors relating them.
  - We don't need fewer schemas; we need to *lower the barrier* to creating and relating schemas.

Summary

- 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.

Status

- 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

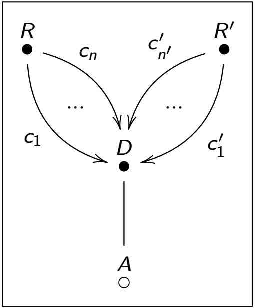
Thanks for inviting me to speak!

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.

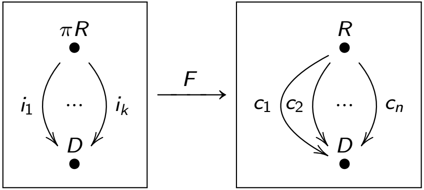
SQL as FQL – the encoding

- 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 $R'(c'_1, \ldots, c'_{n'})$



SQL as FQL – Projection

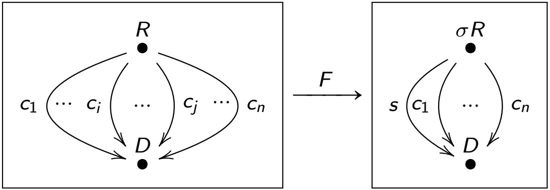
- Let R be a table. We can express $\pi_{i_1,\ldots,i_k} R$ using Δ_F



This construction is only appropriate for bag semantics because πR will will have the same number of rows as R.

SQL as FQL – Selection

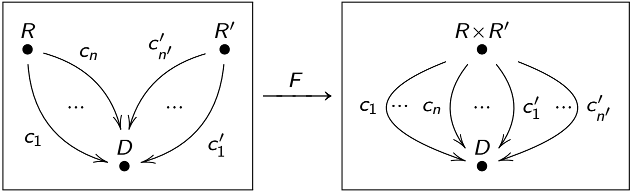
- Let R be a table. We can express $\sigma_{i=j}R$ using Π_F:



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

SQL as FQL – Product

- Let R and R' be tables. We can express $R \times R'$ as Π_F



FQL as SQL – implementing the data migration functors

- Δ 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.
- Σ can be implemented with unions of conjunctive queries and ID-generation.
  - Algorithm is similar to a "union of Δs".
- Π 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.