

# Introduction to Categorical Database Theory

---

Emilio Minichiello

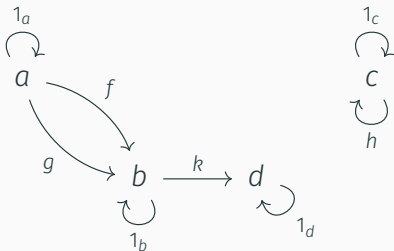
CUNY CityTech

# Categories

---

# Categories

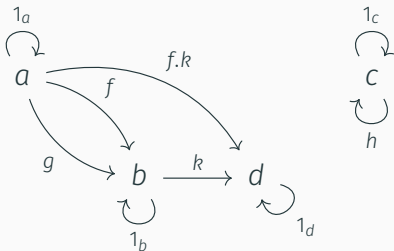
A **category** is a directed multigraph which has a composition operation on arrows.



We call the nodes here **objects**, and we refer to the arrows as **morphisms**. Every object  $u$  has an **identity morphism**  $1_u : u \rightarrow u$ .

# Categories

We can compose morphisms. For example we have  $f: a \rightarrow b$  and  $k: b \rightarrow d$ , so composition gives us a new morphism  $f.k: a \rightarrow d$ .



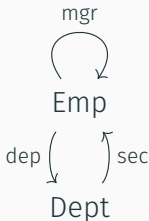
Note: Composition is usually written as  $k \circ f$  rather than  $f.k$ .

Some examples of categories:

- The category **Set**, whose objects are sets and whose morphisms are functions,
- The category **Top**, whose objects are topological spaces and whose morphisms are continuous functions,
- If  $(P, \leq)$  is a partially ordered set, then we can think of it as a category, whose objects are the elements of  $P$  and where there is a unique morphism  $p \rightarrow q$  if and only if  $p \leq q$ .

# Categories

We can also generate categories freely from small amounts of data.

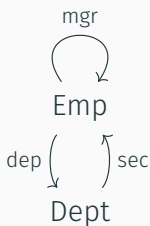


The category above has two objects (Emp and Dept) and infinitely many morphisms

$1_{\text{Emp}}, 1_{\text{Dept}}, \text{mgr}, \text{mgr.mgr}, \text{mgr.mgr.mgr}, \dots,$

$\text{dep}, \text{dep.sec}, \text{dep.sec.dep}, \dots, \quad \text{sec}, \text{sec.mgr}, \text{sec.mgr.dep}, \dots$

We can also quotient by equations between morphisms



$$\text{mgr.mgr} = \text{mgr}$$

$$\text{mgr.dep} = \text{dep}$$

$$\text{sec.dep} = 1_{\text{Dept}}$$

Let  $\mathcal{E}$  denote the resulting category.

Now the category  $\mathcal{E}$  has only finitely many morphisms:

$$[\text{mgr}] = [\text{mgr.mgr}], \quad [\text{dep}], [\text{dep.sec}], [\text{dep.sec.dep}] = [\text{dep}]$$

$$[\text{sec}], [\text{sec.mgr.dep}] = [\text{sec.dep}] = [1_{\text{Dept}}]$$



Given two categories  $\mathcal{C}$  and  $\mathcal{D}$ , a **functor**  $F : \mathcal{C} \rightarrow \mathcal{D}$  is like a graph homomorphism, it assigns objects to objects and morphisms to morphisms in such a way that respects identity morphisms and compositions.

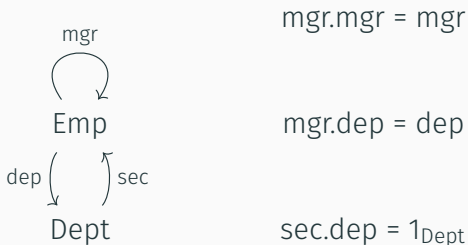
[Spi12] puts forward the following idea:

- Database schemas can be thought of as finitely generated categories  $\mathcal{C}$ , and
- Database instances can be thought of as functors  $I : \mathcal{C} \rightarrow \mathbf{Set}$ .

Let's try and understand the second part of this correspondence.

# Categories

Let us consider the category  $\mathcal{E}$  from before



A functor  $I : \mathcal{E} \rightarrow \mathbf{Set}$  consists of the following data. Sets

$$I(\text{Emp}), \quad I(\text{Dept})$$

and functions

$$I(\text{mgr}) : I(\text{Emp}) \rightarrow I(\text{Emp}), \quad I(\text{dep}) : I(\text{Emp}) \rightarrow I(\text{Dept})$$

$$I(\text{sec}) : I(\text{Dept}) \rightarrow I(\text{Emp})$$

# Categories

How do we think of this as a database?

Let  $I(\text{Emp})$  be the set of employees, and  $I(\text{Dept})$  be the set of departments

$$I(\text{Emp}) = \{\text{Alice}, \text{Bob}, \text{Charlie}\}$$

$$I(\text{Dept}) = \{\text{Math}, \text{CS}\}$$

Then  $I(\text{mgr})$  is a function assigning employees to their managers, and similarly for  $I(\text{dep})$  and  $I(\text{sec})$ .

$$I(\text{mgr}) = (\text{Alice} \mapsto \text{Alice}, \text{Bob} \mapsto \text{Charlie}, \text{Charlie} \mapsto \text{Charlie})$$

$$I(\text{dep}) = (\text{Alice} \mapsto \text{Math}, \text{Bob} \mapsto \text{CS}, \text{Charlie} \mapsto \text{CS})$$

$$I(\text{sec}) = (\text{Math} \mapsto \text{Alice}, \text{CS} \mapsto \text{Charlie})$$

We can summarize all of this information into tables.

Emp	mgr	dep
Alice	Alice	Math
Bob	Charlie	CS
Charlie	Charlie	CS

Dept	sec
Math	Alice
CS	Charlie

The columns dep and sec are usually called **foreign keys** in database theory.

# Data Migration

---

So we have:

- Database Schemas  $\leftrightarrow$  finitely generated categories
- Database Instances  $\leftrightarrow$  functors to **Set**

Now for something new:

- functors between schemas.

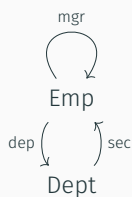
# Data Migration

Consider the following functor

$$\text{mgr}' . \text{mgr}' = \text{mgr}'$$



$$\xrightarrow{F}$$



$$\text{mgr} . \text{mgr} = \text{mgr}$$

$$\text{mgr} . \text{dep} = \text{dep}$$

$$\text{sec} . \text{dep} = 1_{\text{Dept}}$$

$$\mathcal{C} \xrightarrow{F} \mathcal{E}$$

where  $F$  sends  $\text{Emp}'$  to  $\text{Emp}$  and  $\text{mgr}'$  to  $\text{mgr}$ .

If  $I : \mathcal{E} \rightarrow \mathbf{Set}$  is a functor, then we can precompose  $I$  with  $F$  to obtain a new functor

$$\mathcal{C} \xrightarrow{F} \mathcal{E} \xrightarrow{I} \mathbf{Set}.$$



# Data Migration

For example, with  $F$  as above, then for the database  $I : \mathcal{E} \rightarrow \mathbf{Set}$  from before

Emp	mgr	dep
Alice	Alice	Math
Bob	Charlie	CS
Charlie	Charlie	CS

Dept	sec
Math	Alice
CS	Charlie

Precomposing  $I$  with  $F$  gives the functor  $(I \circ F) : \mathcal{C} \rightarrow \mathbf{Set}$

Emp'	mgr'
Alice	Alice
Bob	Charlie
Charlie	Charlie

In other words, by precomposing our database instance  $I$  with the functor  $F$ , we can take a projection of our database.

In fact, this construction defines a functor

$$\mathbf{Set}^{\mathcal{E}} \xrightarrow{\Delta_F} \mathbf{Set}^{\mathcal{C}}$$

it takes a functor  $I : \mathcal{E} \rightarrow \mathbf{Set}$  to a functor  $(I \circ F) : \mathcal{C} \rightarrow \mathbf{Set}$ .

However, this functor  $\Delta_F$  always has left and right adjoint functors, given by Kan extension.

The functor  $\Delta_F$  and its left and right adjoints

$$\begin{array}{ccc} & \Sigma_F & \\ \leftarrow & \text{---} & \rightarrow \\ \text{Set}^{\mathcal{E}} & \Delta_F & \text{Set}^{\mathcal{C}} \\ \leftarrow & \text{---} & \rightarrow \\ & \Pi_F & \end{array}$$

In general,  $\Sigma_F$  and  $\Pi_F$  can be quite complicated to compute. However in special cases, we can understand them fully.

# Data Migration

For example, taking  $F : \mathcal{C} \rightarrow \mathcal{E}$  from before, let  $J : \mathcal{C} \rightarrow \mathbf{Set}$  be the database instance given by

Emp'	mgr'
Arthur	Bailey
Bailey	Bailey
Chuck	Arthur
Dolores	Bailey

Then computing  $\Sigma_F(J)$  gives a database instance on  $\mathcal{E}$ , which looks like

Emp	mgr	dep
Arthur	Bailey	Arthur.dep
Bailey	Bailey	Bailey.dep
Chuck	Arthur	Chuck.dep
Dolores	Bailey	Dolores.dep

Dept	sec
Arthur.dep	Arthur.dep.sec
Bailey.dep	Bailey.dep.sec
Chuck.dep	Chuck.dep.sec
Dolores.dep	Dolores.dep.sec

# Data Migration

This database  $\Sigma_F(J)$  implements a feature known in database theory as **labelled nulls**. In other words, the bold elements below are placeholders, indicating that we don't know what should be put there.

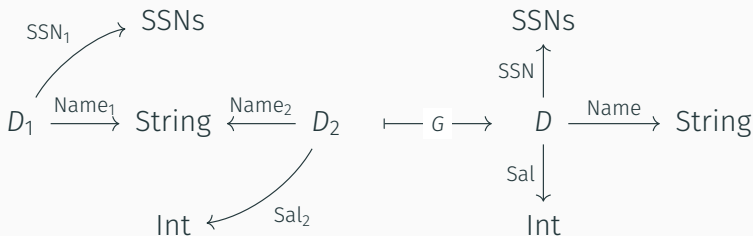
Emp	mgr	dep
Arthur	Bailey	<b>Arthur.dep</b>
Bailey	Bailey	<b>Bailey.dep</b>
Chuck	Arthur	<b>Chuck.dep</b>
Dolores	Bailey	<b>Dolores.dep</b>

Dept	sec
<b>Arthur.dep</b>	<b>Arthur.dep.sec</b>
<b>Bailey.dep</b>	<b>Bailey.dep.sec</b>
<b>Chuck.dep</b>	<b>Chuck.dep.sec</b>
<b>Dolores.dep</b>	<b>Dolores.dep.sec</b>

This functor  $\Sigma_F$  can be computed with what is called the **chase algorithm** in database theory, see [MSW22] for more information on this connection.

# Data Migration

The other functor  $\Pi_F$  acts very similarly to **joins** in database theory. For example, consider the functor  $G : \mathcal{A} \rightarrow \mathcal{B}$



which sends  $D_1, D_2 \mapsto D$ . Suppose we have a database instance  $K$  on  $\mathcal{A}$ :

$D_1$	$Name_1$	$SSN_1$
ID 403	Bailey	123456789
ID 333	Arthur	456321987
ID 123	Chuck	987654321

$D_2$	$Name_2$	$Sal_2$
ID A	Alice	150
ID B	Arthur	100
ID C	Bailey	120
ID D	Dolores	90

# Data Migration

So for the database instance  $K$  on the schema  $\mathcal{A}$

$D_1$	$Name_1$	$SSN_1$
ID 403	Bailey	123456789
ID 333	Arthur	456321987
ID 123	Chuck	987654321

$D_2$	$Name_2$	$Sal_2$
ID a	Alice	150
ID b	Arthur	100
ID c	Bailey	120
ID d	Dolores	90

We obtain a new database instance  $\Pi_G(K)$  on  $\mathcal{B}$

Emp	Name	SSN	Sal
(ID 403, ID c)	Bailey	123456789	120
(ID 333, ID b)	Arthur	456321987	100

This is precisely the join of the two database tables above.

In fact, all of the usual operations one does in SQL can be implemented using categorical operations.

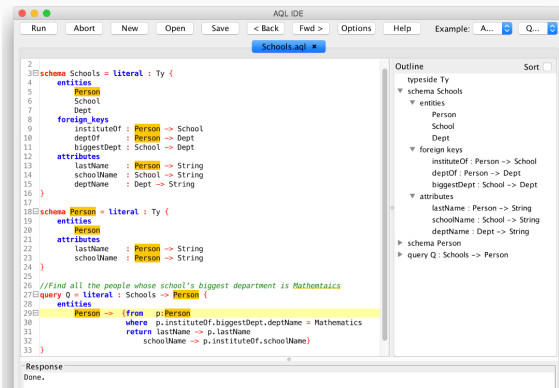
**Theorem** (Wisnesky-Spivak [SW15]): Every SPCU operator (select, projection, cartesian-product and union) can be implemented using compositions of  $\Delta_F, \Sigma_F, \Pi_F$  for appropriate functors  $F$ .



# Data Migration

I worked at a startup company, Conexus, which developed a specialized query programming language called CQL (Categorical Query Language) using this theory.

CONEXUS®



```
2
3 schema Schools = literal : Ty {
4   entities
5     Person
6     School
7     Dept
8   foreign_keys
9     instituteOf : Person -> School
10    deptOf : Person -> Dept
11    biggestDept : School -> Dept
12  attributes
13    lastName : Person -> String
14    schoolName : School -> String
15    deptName : Dept -> String
16 }
17
18 schema Person = literal : Ty {
19   entities
20     Person
21   attributes
22     lastName : Person -> String
23     schoolName : Person -> String
24 }
25
26 //Find all the people whose school's biggest department is Mathematics
27 query Q = literal : Schools -> Person {
28   entities
29     Person -> (from p:Person
30                where p.instituteOf.biggestDept.deptName = Mathematics
31                    return lastName -> p.lastName
32                          schoolName -> p.instituteOf.schoolName)
33 }
```

Response  
Done.

Outline

- typeside Ty
- schema Schools
  - entities
    - Person
    - School
    - Dept
  - foreign keys
    - instituteOf : Person -> School
    - deptOf : Person -> Dept
    - biggestDept : School -> Dept
  - attributes
    - lastName : Person -> String
    - schoolName : School -> String
    - deptName : Dept -> String
- schema Person
  - entity Person
- query Q : Schools -> Person

# Data Migration

The basic idea of Conexus' business model is the following problem:

- Big company has massive database instance  $I$  (many TBs) on schema  $\mathcal{S}$ ,
- Big company decides to change schema  $\mathcal{S}$  to  $\mathcal{S}'$ ,
- database instance  $I$  needs to be migrated to schema  $\mathcal{S}'$ .

This process is sometimes called **database refactoring**. It can be an extremely expensive and error-prone process.

One use of CQL is to verify that this process can go through without errors, ensuring the company doesn't waste hundreds of thousands of dollars.

## Caveats

---

There are two major issues with the story I've told so far:

- we never talked about **attributes**, and
- using  $\Delta, \Sigma, \Pi$  is too unpredictable and computationally difficult to use in practice.

The solutions to these problems are as follows:

- Use **algebraic theories** to model attributes,
- Use **profunctors** to model queries, rather than  $\Delta, \Sigma, \Pi$ .  
Mathematically this is equivalent, but has fascinating computational consequences.

# Caveats

In real life, we want to differentiate between actual data, like 3 or "Emilio" and labelled nulls. In the previous model, there is no distinction, everything is just a set. This leads to the **attribute problem**.

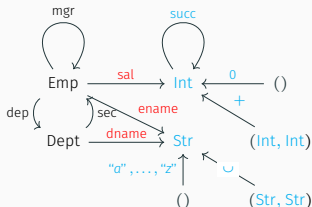
ID	Name	Age	Salary		ID	Name	Age	Salary		ID	Name	Age	Salary
1	Alice	20	\$100	( $\cong$ ) (good)	4	Alice	20	\$100	( $\cong$ ) (bad)	1	Amy	20	\$100
2	Bob	20	\$250		5	Bob	20	\$250		2	Bill	20	\$250
3	Sue	30	\$300		6	Sue	30	\$300		3	Susan	30	\$300

Fig. 3. The Attribute Problem

Figure 1: From [SW17]

# Caveats

Now one needs to separate pieces of schemas into **entities** and **attributes**.



$$e : \text{Emp} \vdash e.\text{mgr}.\text{dep} = e.\text{dep} : \text{Dept}$$

$$e : \text{Emp} \vdash e.\text{mgr}.\text{mgr} = e.\text{mgr} : \text{Emp}$$

$$d : \text{Dept} \vdash d.\text{sec}.\text{dep} = d : \text{Dept}$$

$$e : \text{Emp} \vdash e.\text{mgr}.\text{sal} = e.\text{sal} + 100 : \text{Int}$$

We can also add in new equations.

# Caveats

We can then generate database instances on these schemas using presentations, very similar to group presentations.

**Example:**

Generators:  $(e_0, e_1, e_2 : \text{Emp}, d_0, d_1 : \text{Dept})$

$$\text{Equations: } \left\{ \begin{array}{l} e_0.\text{ename} = \text{"Alice"}, e_1.\text{ename} = \text{"Bob"}, e_2.\text{ename} = \text{"Charlie"}, \\ d_0.\text{dname} = \text{"CS"}, \quad d_1.\text{dname} = \text{"Math"}, \\ e_0.\text{mgr} = e_0, \quad e_1.\text{mgr} = e_2, \quad e_2.\text{mgr} = e_2, \\ \dots \end{array} \right\}$$

Emp	mgr	dep	sal	ename
$e_0$	$e_0$	$d_0$	100	"Alice"
$e_1$	$e_2$	$d_1$	$e_1.\text{sal}$	"Bob"
$e_2$	$e_2$	$d_1$	$e_2.\text{sal}$	"Charlie"

Dept	sec	dname
$d_0$	$e_0$	"CS"
$d_1$	$e_2$	"Math"

This is called the **algebraic data model**, defined and studied in [SW17] and [Sch+17].

This is the actual model of database theory that is implemented in CQL. However, to query a database instance, we use **profunctors**.



# Caveats

**Idea:** A **relation** from a set  $A$  to a set  $B$  is a function  $R : A \times B \rightarrow \mathbf{2}$ . Equivalently a subset  $R \subseteq A \times B$ .

We can compose relations  $R \subseteq A \times B$  and  $S \subseteq B \times C$  with

$$(S \circ R) = \{(a, c) : \exists b \in B \text{ such that } (a, b) \in R \text{ and } (b, c) \in S\}.$$

A **profunctor**  $P : \mathcal{C} \leftrightarrow \mathcal{D}$  is a functor  $P : \mathcal{C}^{\text{op}} \times \mathcal{D} \rightarrow \mathbf{Set}$ .

We can compose profunctors  $P : \mathcal{C} \leftrightarrow \mathcal{D}$ ,  $Q : \mathcal{D} \leftrightarrow \mathcal{E}$  using what is called a coend:

$$(Q \circ P)(c, e) = \int^{d \in \mathcal{D}} P(c, d) \times Q(d, e).$$

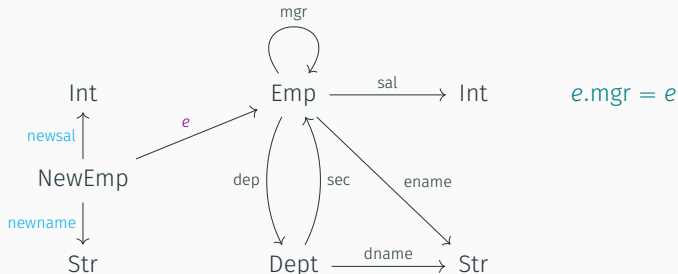
## Caveats

$P$  : “Select the name and (salary + 50) of all employees who are their own manager”

```
SELECT e.ename AS newname, e.sal + 50 AS newsal,
```

```
FROM e : Emp,
```

```
WHERE e.mgr = e;
```



## Caveats

Mathematically, a profunctor  $P : \mathcal{C} \leftrightarrow \mathcal{D}$  is a functor

$$P : \mathcal{C}^{\text{op}} \times \mathcal{D} \rightarrow \mathbf{Set}$$

but this is equivalent to a functor

$$P : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}^{\mathcal{D}}.$$

This process is called **currying**, named after mathematician Haskell Curry.

Strangely enough, while these two descriptions are mathematically equivalent, **they are not computationally equivalent** in a certain precise sense.

## Caveats

This is the content of my most recent paper “Presenting Profunctors”, [RMM24] joint with Gabriel Goren Roig and Joshua Meyers. It was accepted into the proceedings of ACT 2024.

I gave a talk on this paper at the New York City Category Theory Seminar, you can find a video of it [here](#).

We define two different ways to give a “presentation” of a profunctor, inspired by the two equivalent descriptions from before. We call them the **uncurried** and **curried** profunctor presentations.

**Theorem**([RMM24]) Given two finite **curried** profunctor presentations  $P : C \leftrightarrow D$  and  $Q : D \leftrightarrow E$ , then there exists a finite curried presentation  $(Q * P)$  whose semantics  $\llbracket Q * P \rrbracket$  agrees with  $\llbracket Q \rrbracket \circ \llbracket P \rrbracket$ .

However, there exist finite **uncurried** profunctor presentations  $P$  and  $Q$  such that there does not exist a finite uncurried profunctor presentation  $R$  with  $\llbracket R \rrbracket \cong (\llbracket Q \rrbracket \circ \llbracket P \rrbracket)$ .

I think that category theory provides a powerful tool to reason about computational problems, and database theory is one area of computer science where such tools can be very helpful.

Thank you for listening!

Questions?

Comments?

Feel free to [email me](#).

## References

---

- [MSW22] Joshua Meyers, David I Spivak, and Ryan Wisnesky. “Fast left kan extensions using the chase”. *Journal of Automated Reasoning* 66.4 (2022), pp. 805–844.
- [RMM24] Gabriel Goren Roig, Joshua Meyers, and Emilio Minichiello. *Presenting Profunctors*. 2024. arXiv: [2404.01406](https://arxiv.org/abs/2404.01406) [math.CT].
- [Sch+17] Patrick Schultz et al. “Algebraic Databases”. *Theory and Applications of Categories* 32.16 (2017), pp. 547–619.

- [Spi12] David I Spivak. **“Functorial data migration”**. *Information and Computation* 217 (2012), pp. 31–51.
- [SW15] David I Spivak and Ryan Wisnesky. **“Relational foundations for functorial data migration”**. *Proceedings of the 15th Symposium on Database Programming Languages*. 2015, pp. 21–28.
- [SW17] Patrick Schultz and Ryan Wisnesky. **“Algebraic data integration”**. *Journal of Functional Programming* 27 (2017).