

Categorical databases

David I. Spivak

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

Presented on 2012/01/24
at Amgen

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

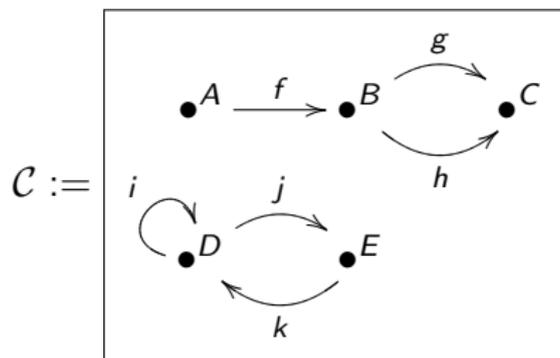
- The connection between databases and categories is simple and strong.
- Reason: categories and database schemas do the same thing.
 - A schema gives a framework for modeling a situation;
 - Tables
 - Attributes
 - This is precisely what a category does.
 - Objects
 - Arrows.
 - They both model how entities within a given context interact.
- Schema = Category.
- In this talk, I'll explain these ideas and some consequences.

Plan of the talk

- Lay out the basic idea of categories and that of databases, and show the tight connection between them.
- Discuss schema evolution and data migration.
- Develop a connection to programming language theory.
- Understand RDF in these terms.

What is a category?

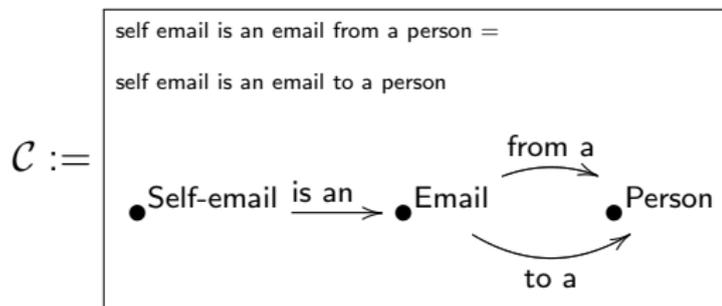
- Idea: A category models entities of a certain sort and the relationships between them.



- Think of it like a graph: the nodes are entities and the arrows are relationships.
- Some paths can be declared equivalent to others
 - Example: declare that $j; k \simeq i; i; i$ and $f; g \simeq f; h$.

Example

- How could one interpret this kind of abstraction?



- Such “business rules” can be encoded into the category.

What is the essence of structure?

- If mathematics is the art of getting organized, what organizes math?
- After thousands of years, people realized that there were some essential features in common throughout much of math.
- These are objects, arrows, paths, and path equivalence.
- Or: things, tasks, processes, and “sameness of outcome”.
- Or: primary keys, foreign keys, paths of FKs, and path equations.

An unusual definition of categories

- The following two slides give a non-standard definition of categories.
- One might call them *ontologies* instead.
- However, the present notion is equivalent (in the category-theoretic sense) to that of categories.
- It seems more informative to call these categories, so that's what I'll do.

Definition of a category I: Constituents

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

- ① A set $\mathbf{Ob}(\mathcal{C})$, called *the set of objects of \mathcal{C}* .
 - (These will be tables.)
 - Objects $x \in \mathbf{Ob}(\mathcal{C})$ is often written as \bullet^x .
- ② A set $\mathbf{Arr}(\mathcal{C})$, called *the set of arrows of \mathcal{C}* , and two functions

$$src, tgt: \mathbf{Arr}(\mathcal{C}) \rightarrow \mathbf{Ob}(\mathcal{C}),$$

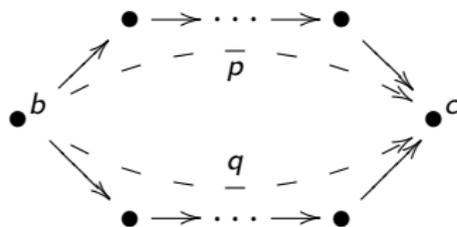
assigning to each arrow its *source* and its *target* object, respectively.

- (Arrows will be foreign keys from "source" table to "target" table.)
 - An arrow $f \in \mathbf{Arr}(\mathcal{C})$ is often written $\bullet^x \xrightarrow{f} \bullet^y$, where $x = src(f), y = tgt(f)$.
 - We define a *path in \mathcal{C}* to be a finite "head-to-tail" sequence of arrows in \mathcal{C} , e.g. $\bullet^x \xrightarrow{f} \bullet^y \xrightarrow{g} \bullet^z$.
- ③ An notion of equivalence for paths, denoted \simeq .

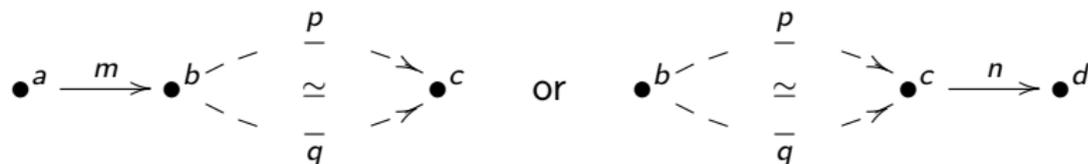
Definition of a category II: Rules

These constituents must satisfy the following requirements:

- 1 If $p \simeq q$ are equivalent paths then the sources agree: $\text{src}(p) = \text{src}(q)$.
- 2 If $p \simeq q$ are equivalent paths then the targets agree: $\text{tgt}(p) = \text{tgt}(q)$.
- 3 Suppose we have two paths (of any lengths) $b \rightarrow c$:



If $p \simeq q$ then for any extensions



$m; p \simeq m; q$

and

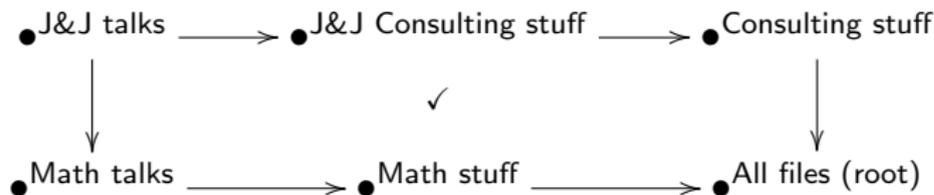
$p; n \simeq q; n$

What does equivalence of paths mean?

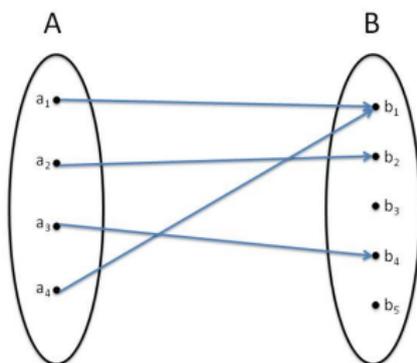
- Arrows represent foreign keys.
- A path $p: \bullet^a \rightarrow \bullet^b$ represents “following foreign keys” from table a to table b .
- Following a path p , we can take any record in table a and return a record in table b .
- We declare two paths $p, q: \bullet^a \rightarrow \bullet^b$ equivalent if they should return the same record in b for any record in a .
- In typical DB practices, equivalent paths are avoided by cutting one of the paths.
 - This is considered good design.
 - However, it often causes pain in ones neck.
 - Category theory has this concept built in.

The power of path equivalences

- Ever wanted two directory paths to contain the same file?
- Example: this “Beamer” presentation was written for a consulting gig at Jack and Jill Inc, and as such belongs both in my math talks folder and in my J&J consulting folder.
- My file system does not allow that, because without path equivalences, it is dangerous.
- With commutative diagrams we can declare two paths equivalent:



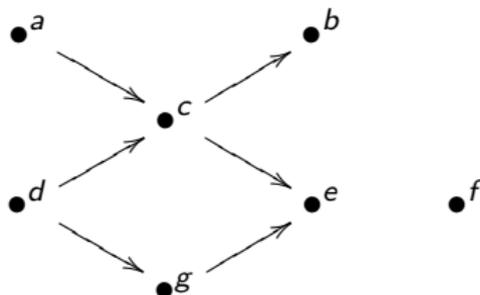
The category of Sets



- Above we see two sets and a function between them. We would denote this categorically by $\bullet^A \xrightarrow{f} \bullet^B$
 - The objects of **Set** represent sets.
 - The arrows in **Set** represent functions.
 - A path represents a sequence of composable functions.
 - Two paths are equivalent if their compositions are the same.
- Note that b_3 and b_5 have been inserted, and a_1 and a_4 have been merged.

A totally different category: an ordered set

- A ordered set is a set S together with a notion of \leq , satisfying
 - $a \leq a$ for all $a \in S$, and
 - if $a \leq b$ and $b \leq c$, then $a \leq c$.
- Given some ordered set S , we can build a corresponding category \mathcal{S} :
 - $\mathbf{Ob}(\mathcal{S}) = S$,
 - One arrow $a \rightarrow b$ if $a \leq b$
 - No arrows $a \rightarrow b$ if $a \not\leq b$.
 - All pairs of paths (having same source and target) are equivalent.
- "Hasse diagram":



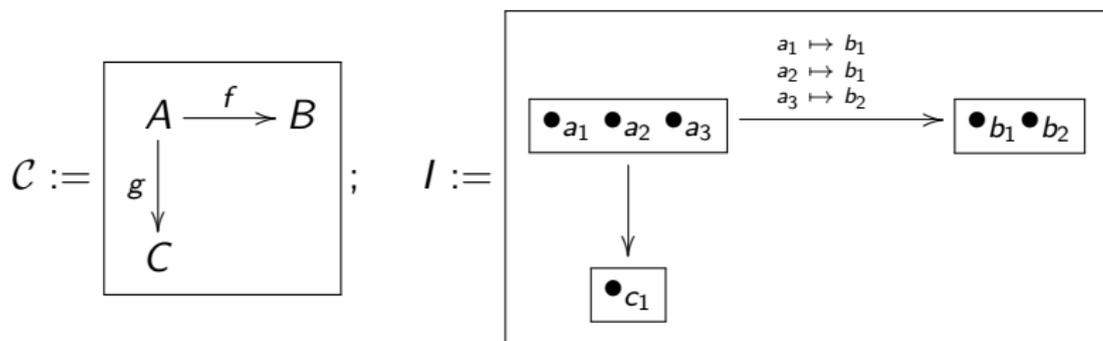
- Think "permissions": $a \leq c$ means a has fewer accessors than b .

Functors: mappings between categories

- One way to think of a category is as a directed graph, where certain paths have been declared equivalent.
- A functor is a graph mapping that is required to respect equivalence of paths.
- **Definition:** A functor $F: \mathcal{C} \rightarrow \mathcal{D}$ consists of
 - a function $\mathbf{Ob}(\mathcal{C}) \rightarrow \mathbf{Ob}(\mathcal{D})$ and
 - a function $\mathbf{Arr}(\mathcal{C}) \rightarrow \mathbf{Path}(\mathcal{D})$,such that F
 - respects sources and targets,
 - respects equivalences of paths.

Functors to **Set**

- A category \mathcal{C} is a system of objects and arrows, and an equivalence relation on its paths.
- A functor $\mathcal{C} \rightarrow \mathcal{D}$ is a mapping that preserves these structures.
- **Set** is the category whose objects are sets, whose arrows are functions, and where paths are equivalent if they compose to the same function.
- If \mathcal{C} is the category on the left below, then a functor $I: \mathcal{C} \rightarrow \mathbf{Set}$ might look like this:



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.”
- An attribute 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 and business rules

- Example:

| Employee | | | | |
|----------|----------|---------|-----|-----|
| Id | First | Last | Mgr | Dpt |
| 101 | David | Hilbert | 103 | q10 |
| 102 | Bertrand | Russell | 102 | x02 |
| 103 | Alan | Turing | 103 | q10 |

| Department | | |
|------------|------------|------|
| Id | Name | Secr |
| q10 | Sales | 101 |
| x02 | Production | 102 |

- Note the Id (primary key) columns and the foreign key columns.
 - Id column could just be internal “row numbers” or could be typed.
 - “Row numbers” (i.e. pointers) are not part of the relational model but they are naturally part of the categorical model.
- 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.
- Examples:

| String |
|-----------|
| Id |
| a |
| b |
| ⋮ |
| z |
| aa |
| ab |
| ⋮ |

| Integer |
|-----------|
| Id |
| 0 |
| 1 |
| ⋮ |
| 9 |
| 10 |
| 11 |
| ⋮ |

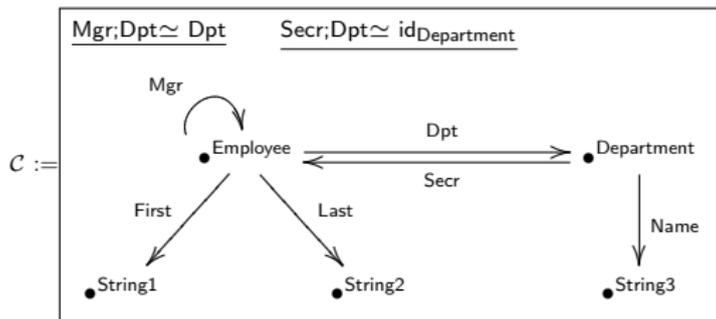
- So even data columns can be considered as foreign keys (to respective 1-column tables).
- Conclusion: each column in a table is a key – one primary, the rest foreign.

Example again

| Employee | | | | |
|----------|----------|---------|-----|-----|
| Id | First | Last | Mgr | Dpt |
| 101 | David | Hilbert | 103 | q10 |
| 102 | Bertrand | Russell | 102 | x02 |
| 103 | Alan | Turing | 103 | q10 |

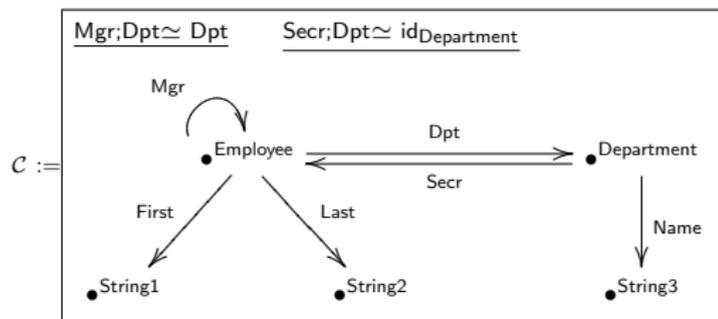
| Department | | |
|------------|------------|------|
| Id | Name | Secr |
| q10 | Sales | 101 |
| x02 | Production | 102 |

| String |
|--------|
| Id |
| a |
| b |
| . |
| . |
| z |
| aa |
| ab |
| . |
| . |
| . |



Database schema as a category

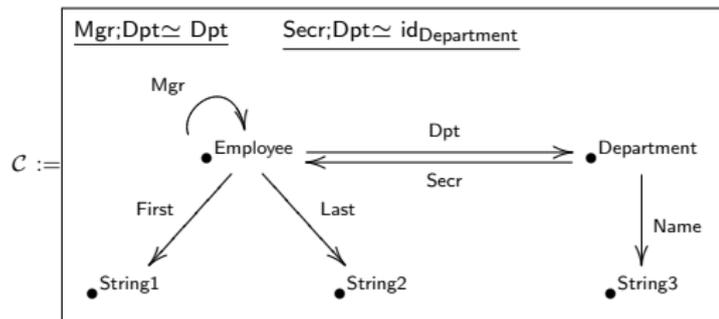
- A database schema is a system of tables linked by foreign keys.
- This is just a category!



- Each object x in \mathcal{C} is a table (Employee, Departments, String);
- each arrow $x \rightarrow y$ is a column of table x .
- Id column of a table corresponds to the trivial path on that object.
- Declaring business rules (e.g. $\text{Mgr}; \text{Dpt} \simeq \text{Dpt}$) is declaring the path equivalence.

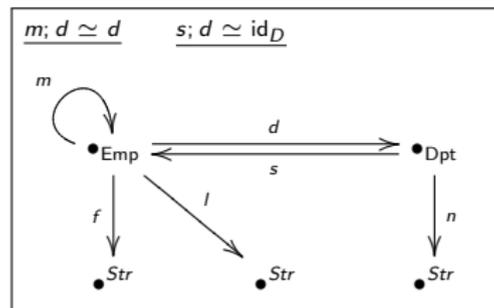
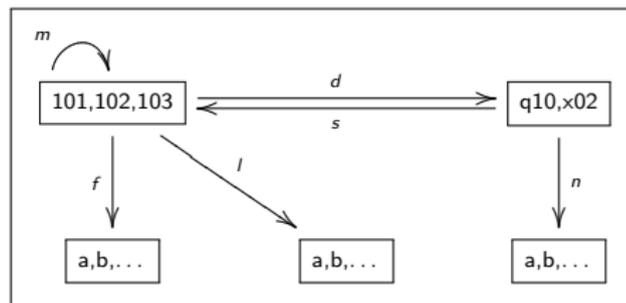
Schema=Category, Instance=Set-valued functor

- Let \mathcal{C} be the following category



- A functor $I: \mathcal{C} \rightarrow \mathbf{Set}$ 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, I fills the schema with compatible data.
- Categorical databases could also be called *functional databases*.

Data as a set-valued functor

 $\mathcal{C} :=$

 $I: \mathcal{C} \rightarrow \mathbf{Set}$


- A category \mathcal{C} is a schema. An object $x \in \mathbf{Ob}(\mathcal{C})$ is a table.
- A functor $I: \mathcal{C} \rightarrow \mathbf{Set}$ fills the tables with compatible data.
- For each table x , the set $I(x)$ is its set of rows.
- The path equivalences in \mathcal{C} are enforced by I as business rules.

Summary

- The connection between categories and databases is simple.
- A schema is a custom category.
- Functors $I: \mathcal{C} \rightarrow \mathbf{Set}$ are instances.
- What about functors $F: \mathcal{C} \rightarrow \mathcal{D}$ between schemas?

Changes

- We've discussed the situation as though static: a single schema and a single instance.
- Next we'll discuss changes.
- Changing the schema (schema mappings).
- Changing the data (updates).

Changes in schema

- Suppose in our modeling of a given context, we evolve from schema \mathcal{C} to schema \mathcal{D} .
- We should find a functorial connection between them.
- Over time we may have something like

$$\mathcal{C} = \mathcal{C}_0 \xrightarrow{F_0} \mathcal{C}_1 \xrightarrow{F_1} \dots \xrightarrow{F_n} \mathcal{C}_n = \mathcal{D}$$

- We want to be able to migrate data from \mathcal{C} to \mathcal{D} and vice versa.
- We want to be able to migrate queries against \mathcal{C} to queries against \mathcal{D} and vice versa.
- And we want this all to work as it “should”.

Composing functors

- Suppose $F: \mathcal{C} \rightarrow \mathcal{D}$ and $G: \mathcal{D} \rightarrow \mathcal{E}$ are functors.
- What is their composition $\mathcal{C} \rightarrow \mathcal{E}$?
 - We have a way to take objects in \mathcal{C} to objects in \mathcal{E} ,
 - Arrows in \mathcal{C} turn into paths in \mathcal{D} and arrows in \mathcal{D} turn into paths in \mathcal{E} .
 - We can concatenate these, thus taking arrows in \mathcal{C} to paths in \mathcal{E} .
 - Our rules ensure that the equivalences in \mathcal{C} will be preserved in \mathcal{E} .
- Composing functors is going to make migrating data more straightforward.

Changes in data

- Let \mathcal{C} be a schema and let $I, J: \mathcal{C} \rightarrow \mathbf{Set}$ be two instances.
- A *natural transformation* $u: I \rightarrow J$ consists of the following:
 - For each object (table) $T \in \mathbf{Ob}(\mathcal{C})$ we get a map of record sets

$$u_T: I(T) \rightarrow J(T).$$

- For each arrow (foreign key) $f: T \rightarrow T'$, we get data consistency; formally,

$$J(f) \circ u_T = u_{T'} \circ I(f).$$

- If J is the result of an insert or merge (a *progressive update*) to I then

$$u: I \rightarrow J.$$

- Same thing if I is the result of a delete or a split (a *regressive update*) to J .

The category of instances

- Given a schema \mathcal{C} , the *category of instances* on \mathcal{C} is denoted $\mathcal{C}\text{-Set}$.
 - The objects of $\mathcal{C}\text{-Set}$ are functors (instances) $I: \mathcal{C} \rightarrow \mathbf{Set}$.
 - The arrows are natural transformations (progressive updates).
 - The paths are sequences of progressive updates.
 - Two paths are equivalent if they result in the same mapping.
- The category $\mathcal{C}\text{-Set}$ is a topos; it has an internal language and logic supporting the *typed lambda calculus*.
- That means, it works well with the theory of programming languages.

Data migration

- Let \mathcal{C} and \mathcal{D} be different schemas.
- We call a functor between them, $F: \mathcal{C} \rightarrow \mathcal{D}$, a *schema mapping*.
- Given such a mapping, we want to be able to canonically transfer instances on \mathcal{C} to instances on \mathcal{D} and vice versa.
- That means, given $F: \mathcal{C} \rightarrow \mathcal{D}$ we want functors

$$\mathcal{C}\text{-Set} \rightarrow \mathcal{D}\text{-Set}$$

and

$$\mathcal{D}\text{-Set} \rightarrow \mathcal{C}\text{-Set}.$$

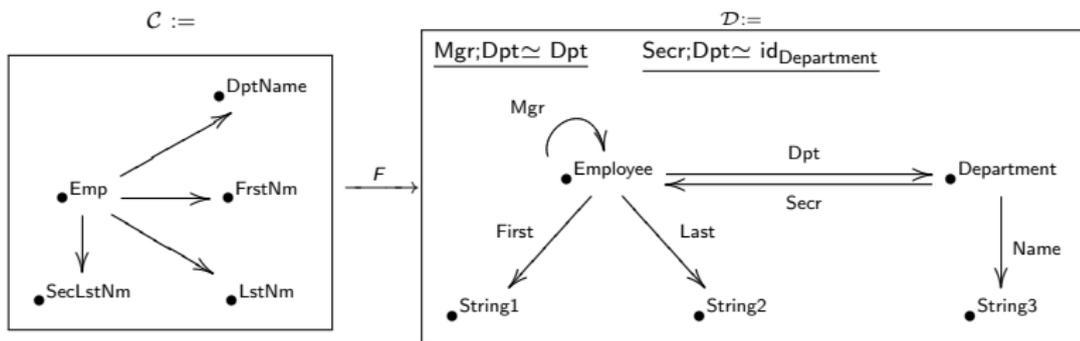
What a functor $\mathcal{C}\text{-Set} \rightarrow \mathcal{D}\text{-Set}$ means.

A functor $\mathcal{C}\text{-Set} \rightarrow \mathcal{D}\text{-Set}$ means:

- **Objects:** To every instance on \mathcal{C} we associate an instance on \mathcal{D} .
- **Arrows:** For every progressive update on a \mathcal{C} -instance there is a corresponding progressive update on the associated \mathcal{D} -instance.
- **Path equivalences:** If two different sequences of progressive updates on \mathcal{C} -instances result in the same mapping, then the same will hold of the corresponding sequences on \mathcal{D} -instances.

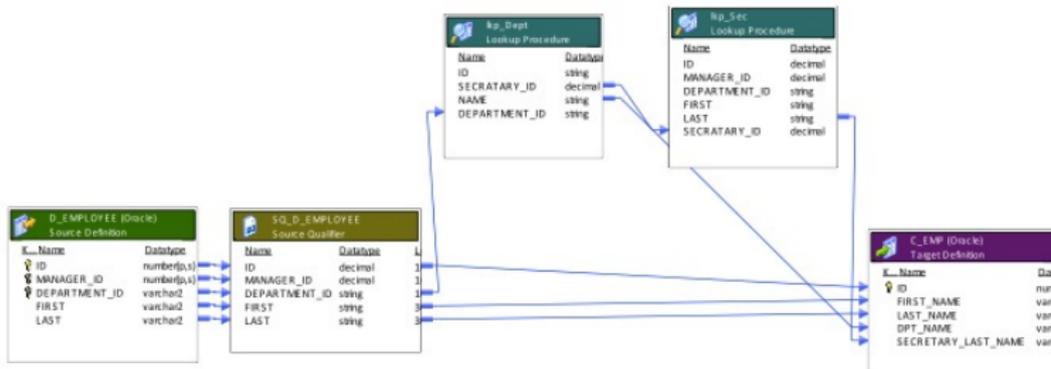
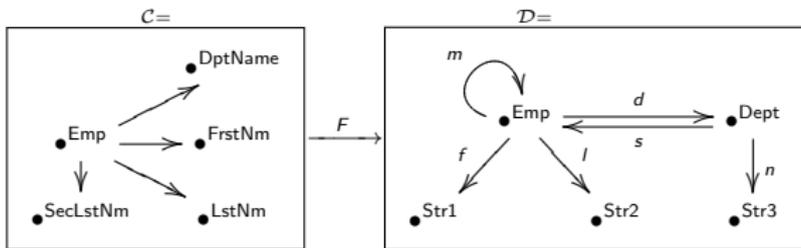
How Δ_F works

- Consider the schema mapping



- We get $\Delta_F: \mathcal{D}\text{-Set} \rightarrow \mathcal{C}\text{-Set}$
- Given an instance on \mathcal{D} we get one on \mathcal{C} .
- Given an update on \mathcal{D} we get one on \mathcal{C} .

Compare the Informatica picture



So many kinds of functors..

- Functors in three different contexts.
 - We started with functors as instances, $I: \mathcal{C} \rightarrow \mathbf{Set}$.
 - Then we introduced functors as schema mappings, $F: \mathcal{C} \rightarrow \mathcal{D}$.
 - In the last slide we showed a functor on instance categories

$$\Delta_F: \mathcal{D}\text{-Set} \rightarrow \mathcal{C}\text{-Set}.$$

- Recall the simple definition of functor we gave at the beginning: it holds in each case.
- Functors provide a powerful and reusable abstraction because of the simplicity of their definition.

Adjoints

- Some functors $\mathcal{X} \rightarrow \mathcal{Y}$ have a “special partner” $\mathcal{Y} \rightarrow \mathcal{X}$ called an *adjoint*.
- What it will mean to us is that we can always “invert” a data migration $\mathcal{D}\text{-Set} \rightarrow \mathcal{C}\text{-Set}$ in two universal ways.
 - Roughly, our first inversion will be universal for progressive updates.
 - Our second inversion will be universal for regressive updates.
- These migration functors will provide something like updatable views.
- The important thing to note is that these aren’t made up; they are “canonical” or “universal”. They’re part of the mathematics – they come with the package.

The “adjoint” migration functors, Σ and Π

Given a schema mapping (i.e. a functor) $F: \mathcal{C} \rightarrow \mathcal{D}$,

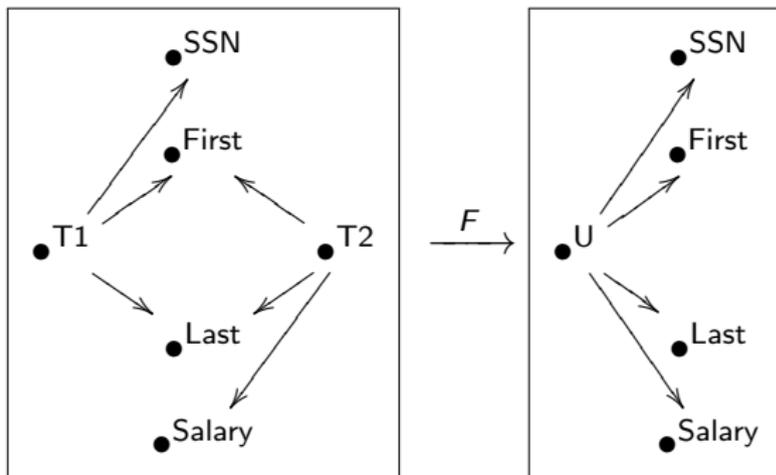
- We have a functor $\Delta_F: \mathcal{D}\text{-Set} \rightarrow \mathcal{C}\text{-Set}$ given by composition.
- It has two adjoints:
 - a “sum-oriented” adjoint $\Sigma_F: \mathcal{C}\text{-Set} \rightarrow \mathcal{D}\text{-Set}$, and
 - a “product-oriented” adjoint $\Pi_F: \mathcal{C}\text{-Set} \rightarrow \mathcal{D}\text{-Set}$.
- Thus, given a schema mapping F , three functors emerge for the instance categories,

$$\Delta_F, \Sigma_F, \text{ and } \Pi_F$$

come with the package.

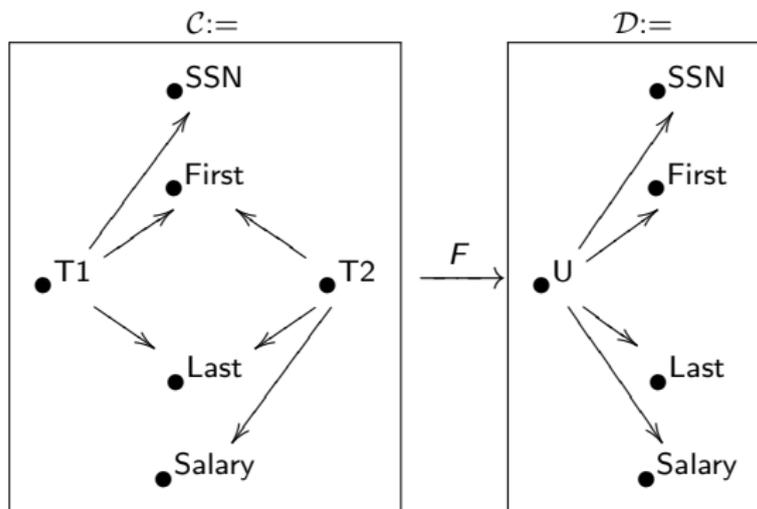
- Roughly, these correspond to project (Δ), union (Σ), and join (Π).
- They allow one to move data back and forth between \mathcal{C} and \mathcal{D} in canonical ways.

The “product-oriented” push-forward Π_F makes joins



- Given any instance $I: \mathcal{C} \rightarrow \mathbf{Set}$, get an instance $\Pi_F(I): \mathcal{D} \rightarrow \mathbf{Set}$.
- The rows in table \bullet^U will be the join of the rows in \bullet^{T1} and \bullet^{T2} over \bullet^{First} and \bullet^{Last} .

The “sum-oriented” push-forward Σ_F makes unions



- Given any instance $I: \mathcal{C} \rightarrow \mathbf{Set}$, get an instance $\Sigma_F(I): \mathcal{D} \rightarrow \mathbf{Set}$.
- The rows in table \bullet^U will be the union of the rows in \bullet^{T1} and \bullet^{T2} .
- It will automatically use labeled nulls for the unknown cells.

Views

- These functors can be arbitrarily composed to create views.
- We can think of any series of functors

$$\mathcal{C}_1 \xleftarrow{F_1} \mathcal{D}_1 \xrightarrow{G_1} \mathcal{E}_1 \xrightarrow{H_1} \mathcal{C}_2 \xleftarrow{F_2} \mathcal{D}_2 \xrightarrow{G_2} \dots \xrightarrow{H_n} \mathcal{C}_n$$

as a view.

- The view is the functor

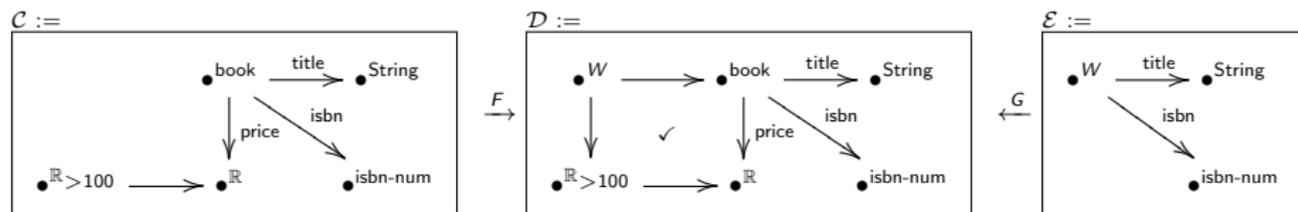
$$V := \Sigma_{H_n} \circ \dots \circ \Pi_{G_1} \circ \Delta_{F_1} : \mathcal{C}_1\text{-Set} \rightarrow \mathcal{C}_n\text{-Set}.$$

- We can export data from \mathcal{C}_1 into \mathcal{C}_n through V .
- Note that \mathcal{C}_n is a schema: not just one table, but possibly many, with foreign keys.
- It's no problem to create views that have foreign keys (unsupported in DBMS).

A simple “SELECT” query using views

```

SELECT title, isbn
FROM book
WHERE price > 100
    
```



- $V := \Delta_G \circ \Pi_F$ is the appropriate view.
- For any $I: \mathcal{C} \rightarrow \mathbf{Set}$, we materialize the view as $V(I)$.
- Views with foreign keys are easy.

One more slide about views

- Views can look complex.
 - We can think of any series of functors

$$\mathcal{C}_1 \xleftarrow{F_1} \mathcal{D}_1 \xrightarrow{G_1} \mathcal{E}_1 \xrightarrow{H_1} \mathcal{C}_2 \xleftarrow{F_2} \mathcal{D}_2 \xrightarrow{G_2} \dots \xrightarrow{H_n} \mathcal{C}_n$$

as describing a view.

- In actuality, the view is the functor

$$V := \Sigma_{H_n} \circ \dots \circ \Pi_{G_1} \circ \Delta_{F_1} : \mathcal{C}_1\text{-Set} \rightarrow \mathcal{C}_n\text{-Set}.$$

- We can materialize the view for any $I : \mathcal{C}_1 \rightarrow \mathbf{Set}$ as $V(I) : \mathcal{C}_n \rightarrow \mathbf{Set}$.
- But a theorem says we can accomplish the same thing in three steps:

$$\mathcal{C}_1 \xleftarrow{F} \mathcal{D} \xrightarrow{G} \mathcal{E} \xrightarrow{H} \mathcal{C}_n$$

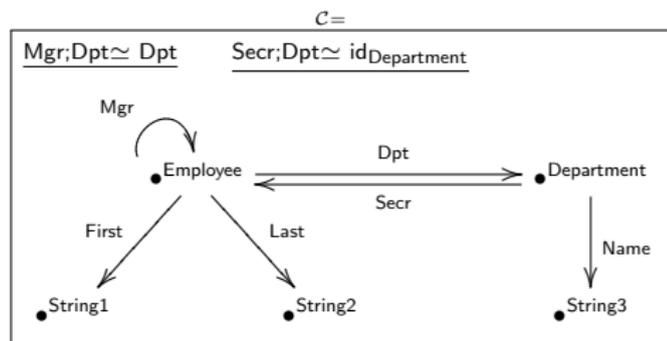
- Project – Join – Union.

Interfacing between schemas

- 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 our notion of views.
- Queries on the old schema translate directly to queries on the new schema.
- We might need to perform calculations such as concatenation, addition, comparison, conversion of units, etc. in order to interface these schemas.
- To do this we'll need an underlying “typing category.”

Incorporating data types and functions

- In the example:



how do we know that \bullet^{String} is what it says it is?

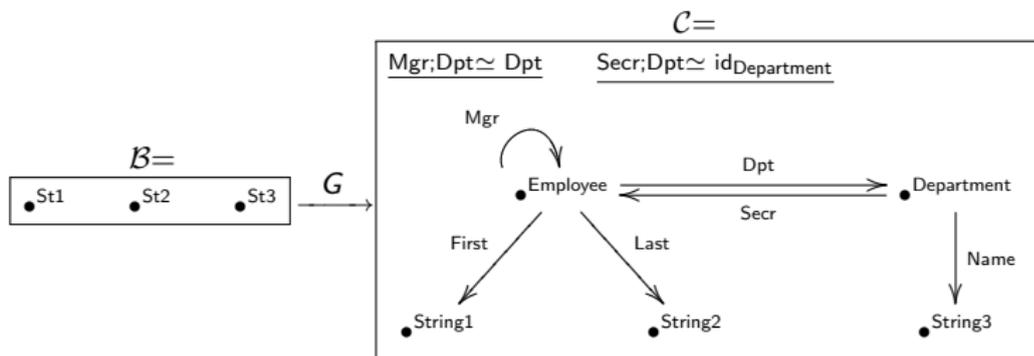
- That is, given $I: \mathcal{C} \rightarrow \mathbf{Set}$, how do we specify that $I(\bullet^{\text{String}}) \in \mathbf{Set}$ is some pre-defined data type like **String**.

Power of category theory: connection to PL is easy

- In programming language theory, they consider the category **Type**.
 - Objects of **Type** are data types, and
 - arrows are functions.
 - Theoretically, there exists a functor $V: \mathbf{Type} \rightarrow \mathbf{Set}$.
- So **Type** is (in our definition) a database schema and V is a “canonical instance”!
- Since database schemas are categories and **Type** is a category, we can integrate the two.

Example

- Lets make a category $\mathcal{B} = \boxed{\bullet \text{St1} \quad \bullet \text{St2} \quad \bullet \text{St3}}$ and a functor $F: \mathcal{B} \rightarrow \mathbf{Type}$, sending each object to $\mathbf{String} \in \mathbf{Ob}(\mathbf{Type})$.
- The composition $\mathcal{B} \xrightarrow{F} \mathbf{Type} \xrightarrow{V} \mathbf{Set}$ yields an instance $V' := \Delta_F(V) = V \circ F: \mathcal{B} \rightarrow \mathbf{Set}$.
- There is also an obvious functor



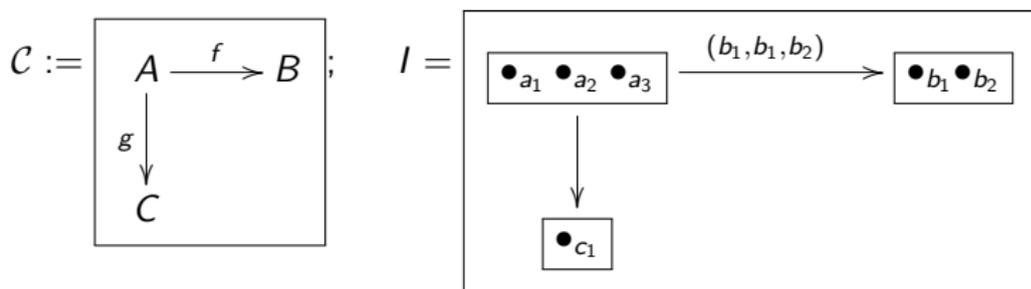
- A typed instance $I: \mathcal{C} \rightarrow \mathbf{Set}$ is one for which we have a map $\Delta_G(I) \rightarrow V'$.

Takeaway

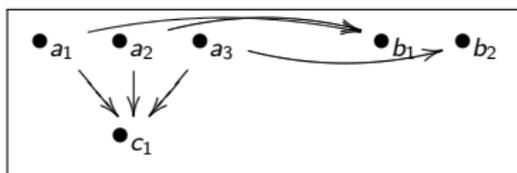
- Databases are custom categories.
- The datatypes in a programming language form a category.
- The whole point of category theory is to allow us to connect different categories.
- Unifying database and program could be very beneficial.

The Grothendieck construction

- Let \mathcal{C} be a category and let $I: \mathcal{C} \rightarrow \mathbf{Set}$ be a functor.
- We can convert I into a category $Gr(I)$ in a canonical way:
 - Example:



- $Gr(I)$ is also known as *the category of elements of I* :

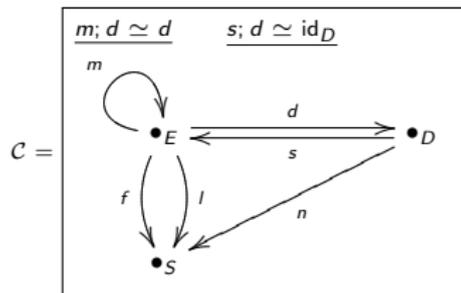


Grothendieck construction applied to database instances

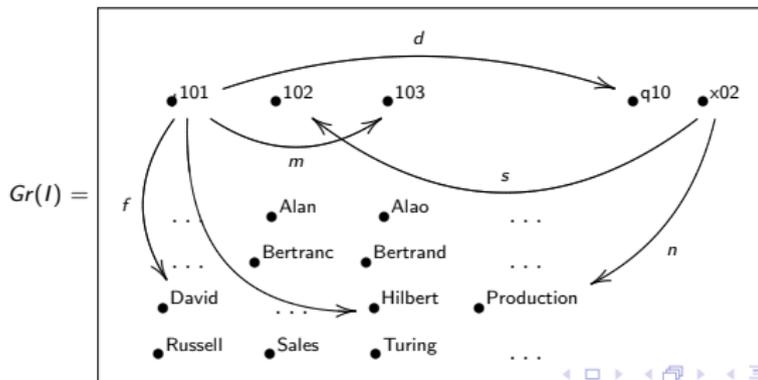
- Suppose given the following instance, considered as $I: \mathcal{C} \rightarrow \mathbf{Set}$

| Employee | | | | |
|----------|----------|---------|-----|-----|
| Id | First | Last | Mgr | Dpt |
| 101 | David | Hilbert | 103 | q10 |
| 102 | Bertrand | Russell | 102 | x02 |
| 103 | Alan | Turing | 103 | q10 |

| Department | | |
|------------|------------|--------|
| Id | Name | Secr'y |
| q10 | Sales | 101 |
| x02 | Production | 102 |



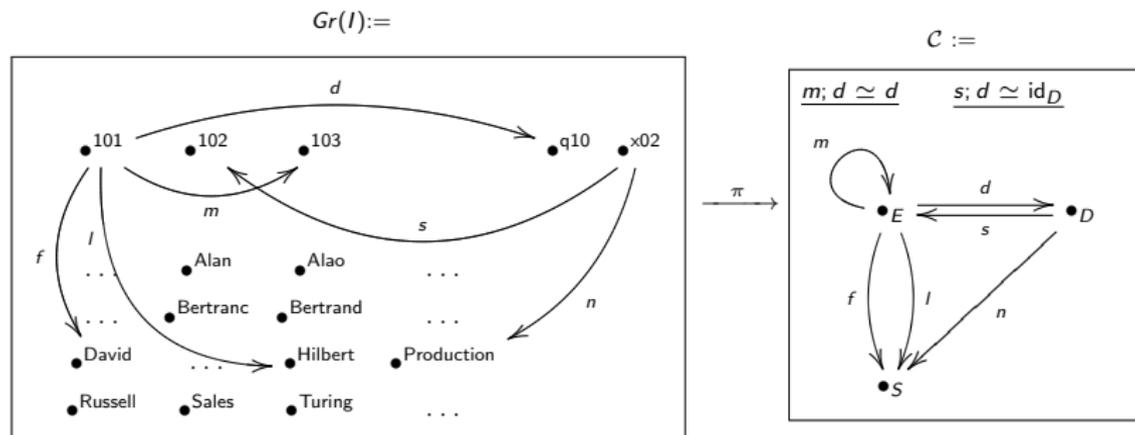
Here is $Gr(I)$, the category of elements of I :



A different perspective on data

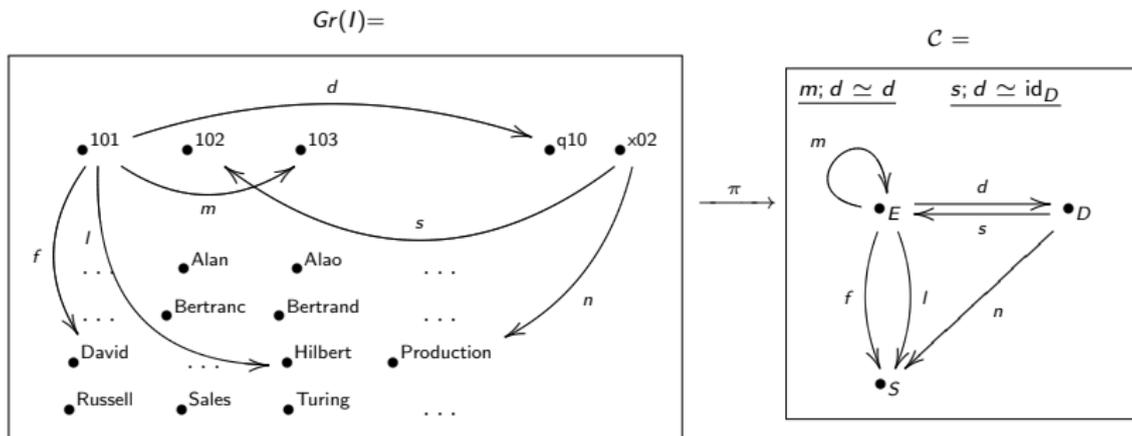
In fact, the Grothendieck construction of $I: \mathcal{C} \rightarrow \mathbf{Set}$ always yields not only a category $Gr(I)$ but a functor

$$\pi: Gr(I) \rightarrow \mathcal{C}.$$



The fiber over (inverse image of) every object $X \in \mathcal{C}$ is a set of objects $\pi^{-1}(X) \subseteq Gr(I)$. That set is $I(X)$.

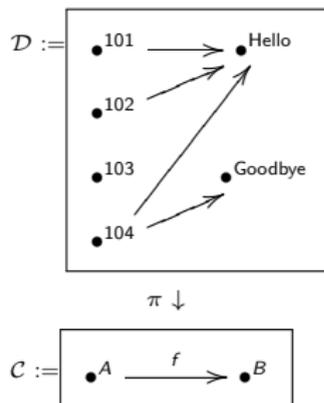
RDF schema and stores



- The relation to RDF triples is clear: each arrow $f: x \rightarrow y$ in $Gr(I)$ is a triple with subject x , predicate f , and object y .
- For example (101 department q10), (x02 name Production), etc..
- \mathcal{C} is the RDF schema and $Gr(I)$ is the triple store.
- SPARQL queries (graph patterns) are easily expressible in this model.

Allowing for semi-structured data

- We can think of any functor $\pi: \mathcal{D} \rightarrow \mathcal{C}$ as a “semi-instance” on \mathcal{C} .
- Such a functor π 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 path equivalences) can also occur in a functor $\pi: \mathcal{D} \rightarrow \mathcal{C}$.
- Any semi-instance on \mathcal{C} can be functorially “corrected” to an instance if necessary.
- For example “labeled nulls” will be created for any incomplete data.

Summary

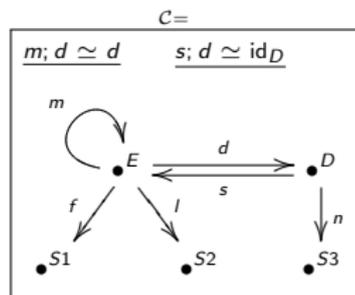
- There's a well-known connection between relational databases and RDF.
- This connection is born out in a most natural way with category theory.
- The model gracefully extends – what should work works.

Summary of the talk

- I hope the connection between databases and categories is clear.

| Employee | | | | |
|----------|----------|---------|-----|-----|
| Id | First | Last | Mgr | Dpt |
| 101 | David | Hilbert | 103 | q10 |
| 102 | Bertrand | Russell | 102 | x02 |
| 103 | Alan | Turing | 103 | q10 |

| Department | | |
|------------|------------|------|
| Id | Name | Secr |
| q10 | Sales | 101 |
| x02 | Production | 102 |



- I discussed how one can use this connection to facilitate:
 - schema mapping and data migration;
 - formalizing views;
 - merging database and programming language theory;
 - merging relational and RDF outlooks;
- The main point is that basic category theory provides a self-contained, unified, and profitable approach to databases.

Thanks for the invitation to speak!