Category Theory as a Unifying Database Formalism

ABSTRACT

Database theory developed from mathematical logic and set theory, but many formalisms have been introduced over time to tackle specific extensions of these core theories. This paper makes the case for using category theory as a unifying formalism for databases, by arguing that category theory can capture in a uniform and succinct way the basic relational model (e.g., schemas, constraints, queries, updates), and its many extensions (e.g., aggregates, transactions, triggers, schema mapping, pivoting, anonymization, and semi-structured data). We also show how existing results from the large corpus of theorems developed within pure mathematics can be naturally imported, and how automated proof assistants such as COQ could be leveraged. Finally, we observe that category theory has been successfully applied to other areas of computer science (e.g., programming languages), this provides hope about the feasibility of our attempt, and suggests that by sharing a rigorous formal footing we could enable fascinating cross-area analysis and theories.

1. INTRODUCTION

DS: Much of the theory below would work out if we used only finite schemas (only a finite number of non-equivalent paths) and allowed only finitely many rows in each table. Would you prefer this? If so, should we still call the category of states on $C^{	ext{Fin}}$ or should we replace $\text{Set}$ with something like $\text{Fin}$ or $\text{FSet}$?

The field of Database research developed at the boundary of active industrial development and solid formal theories, and here more than in any other field this duality has produced a fascinating interplay between system builders and theoreticians. The flow of innovation has not always been a unidirectional transfer from new advances in theory to corresponding systems, but rather an intricate back and forth, where often systems have been built before a full understanding was available.

The core of database theory has developed from rather old and well-understood mathematics: mathematical logic and set theory. From this common starting point the database community has built a host of extensions, which were needed to provide the formal underpinnings for a large number of new functionalities and new solutions to problems arising. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Copyright 200X ACM X-XXXX-XX-XX/XX ...$5.00.

from the system development side. For example consider the various formal advances regarding schema mapping, aggregates and user-defined-functions, deductive databases, incomplete data and management of nulls, database recovery and concurrency control, real-time DB. DS: are we proposing to handle each of these?

While for each of these areas precise formal models, rooted in formal logic, have been provided, it remains often hard to precisely understand the correspondences between different formalizations — at the very least this understanding requires a significant time investment. This task is often daunting, making it harder to operate across the boundaries of different theories, which in turn works as a disincentive to anyone trying to reduce this gap.

A similar problem faced the mathematics community in the first half of the 20th century. Mathematics was subdivided into many subfields, each with its own jargon and way of doing business. In order to advance, a unifying language and formalism was needed. Category theory was invented in the 1940s by Mac Lane and Eilenberg to strengthen the connection between topology and algebra, but it quickly spread to neighboring fields. By providing a precise language for comparing different universes of discourse, category theory has been a unifying force for mathematics.

CC: next two paragraphs need smoothing Category theory is no stranger to computer science. It has been remarkably successful in formalizing the semantics of programming languages[]. In fact, it has also been applied many times to database theory in the past. These attempts did not catch on in mainstream applications, perhaps because those models attempted to be too faithful to the relational model. Because these formalisms did not seem to offer enough advantages to justify the learning curve, some database theorists developed a slight aversion to category theory. CC: We need to check who's in the PC, and avoid to make enemies!

However in this paper, we argue that database theory and category theory are naturally compatible, in fact that a basic database schema and a category are roughly the same thing. Once a simple dictionary is set up, classical category theoretic results and constructions capture many of the formal results that appear in database literature.

In particular, we present an application of categories and functors to uniformly represent schemas, integrity constraints, database states, queries, updates, and show that this is not only natural but enlightening. And we show, with a simple online demo, that simple algorithms can be used to translate SQL schemas (with primary keys and foreign keys) and queries into the formalism of categories and functors.
Next, we showcase the expressivity of category theory in modeling classical database problems, many of which required significant extensions of the relational model, such as schema mapping, transactions, and user-defined aggregates. We show existing results in DB theory can be proved using category theories.

We then argue that by adopting this formalism we can also: (i) inherit results from the large corpus of theorems produced in the pure category theory research, and (ii) leverage automated proof assistants such as COQ.

Finally, since category theory has already been embraced by other communities such as the theoretical programming language community, there is an interesting opportunity of bridging results and enabling theories and analysis to span across different areas. As an example consider security models. As an example the semantics of one kind of attacks can be proved using databases to anyone familiar with computer science. Database results in category theory, e.g., functors composition, perfectly fit known theorems in database theory, e.g., category of sets, can safely be proved in this new framework, Section 5 shows how some of the classical database results can be proved in this new framework, Section 6 provides summary of related work, and Section ?? summarizes our conclusions.

1.0.1 Acknowledgments

The first author would like to thank Peter Gates for many useful conversations and Greg Morrisett for suggesting we explore transactions.

2. BACKGROUND

This section provides the reader with a crash course in category theory. More details can be found in [7, 8, 9, 10]. The reader already familiar with the definition of category and functor, as well as the category Set of sets, can safely skip to Section ?? The goal of this section is to provide database researchers with a core introduction to category theory, so we slightly abuse our notation and postpone detailed comments on some set-theoretic issues (e.g., the distinction between sets and classes) to Appendix B.

Intuitively, a category is a multi-graph (i.e., a graph that can have multiple edges between the same two nodes) together with an equivalence relation on finite paths (i.e., we can declare two paths between the same nodes to be equivalent or not) — as shown in the example figure below.

Each node represents an object of a specific type, each arrow \( A \rightarrow B \) represents a chosen function between objects (that can be total, continuous, etc.), a path in the graph represents the composition of such functions, and two paths are equivalent if they represent the same function. Conceptually, the graph contains the transitive closure of every finite path, including the paths of length 0, i.e. the degenerate self-loop on each node (\( a, b, c \) in our example). Note that we often omit obvious paths when representing a category pictorially. A more formal definition is presented below:

**Definition 2.0.1.** A category \( C \) consists of the following components:

1. A set of objects \( \text{Ob}_C \) — each element \( x \in \text{Ob}_C \) is called an object of \( C \).
2. For each \( x, y \in \text{Ob}_C \) a set \( \text{Arr}_C(x,y) \) — each element \( f \in \text{Arr}_C(x,y) \) is called an arrow from \( x \) to \( y \) in \( C \), and is denoted as \( f : x \rightarrow y \) or as \( x \Rightarrow y \).
3. For each object \( x \in \text{Ob}_C \) a chosen arrow \( \text{id}_x \in \text{Arr}_C(x,x) \) called the identity arrow on \( x \).
4. For each \( x, y, z \in \text{Ob}_C \), a function

\[
\text{comp}_C : \text{Arr}_C(x,y) \times \text{Arr}_C(y,z) \rightarrow \text{Arr}_C(x,z)
\]

called the composition law for \( C \) — we denote \( \text{comp}_C(f,g) \) simply by \( f \circ g \).

To be a category the above components must also satisfy the following properties:

**Identity law:** For each arrow \( f : x \rightarrow y \), the following equations hold:

\[
\text{id}_x \circ f = f \quad \text{and} \quad f \circ \text{id}_y = f.
\]

**Associative law:** Given a sequence of composable arrows

\[
w \xrightarrow{f} x \xrightarrow{g} y \xrightarrow{h} z,
\]

the following equation holds:

\[
f \circ (g \circ h) = (f \circ g) \circ h.
\]
CC: I pulled this triangle comment from the definition.

Given the following triangle:

```
  x   f   y
  |   |   |
  h   g   z
```

The composition commutes iff \( f \circ g = \text{comp}_C(f, g) = h \).

We don’t like using bullet (•) both to denote the composition law and to draw objects in a category (as •). How about \( f \ast g \) for composition?

CC: Let’s use an empty bullet in the graphs, and keep the solid one for the text.

Mathematicians often use categories to formalize rather general concepts such as the category of topological spaces and continuous functions, the category of groups and group homomorphisms, or the category \( \text{Set} \) of sets and functions. The last of these, \( \text{Set} \), is an important category both in mathematics and for the present paper. The objects in this category are sets \(^1\), the arrows are (total) functions between sets, and the composition law simply sends a pair of composable functions to their composition. In this paper we are not interested platonic categories (those of mathematical interest), other than the \( \text{Set} \) category and various derivations of it. Instead we deal mainly with “custom” categories designed to model some enterprise. Our objects will represent tables, our arrows will represent attributes, and our composition law will represent integrity constraints or “business rules.” All this will be made explicit in Section 3.

Another key notion is the one of functors. A functor is a mapping between categories that sends objects to objects, and arrows to arrows, while preserving compositions and identities. We formalize the notion of functor as follows:

**Definition 2.0.2.** Let \( \mathcal{C} \) and \( \mathcal{D} \) be categories. A functor from \( \mathcal{C} \) to \( \mathcal{D} \), denoted \( F: \mathcal{C} \rightarrow \mathcal{D} \), maps:

1. each object \( x \in \text{Ob}_\mathcal{C} \) to an object \( F(x) \in \text{Ob}_\mathcal{D} \), and
2. each arrow \( f: x \rightarrow y \) in \( \text{Arr}_\mathcal{C}(x, y) \) to an arrow \( F(f): F(x) \rightarrow F(y) \) in \( \text{Arr}_\mathcal{D}(F(x), F(y)) \).

A functor must also satisfy the following requirements:

**Preserve identities:** for each object \( x \in \text{Ob}_\mathcal{C} \) the equation \( F(\text{id}_x) = \text{id}_{F(x)} \) holds in \( \text{Arr}_\mathcal{D}(F(x), F(x)) \), and

**Preserve composition:** for each composable pair of arrows \( x \rightarrow y \rightarrow z \) in \( \mathcal{C} \), the equation \( F(f \circ g) = F(f) \circ F(g) \) holds in \( \text{Arr}_\mathcal{D}(F(x), F(z)) \).

Specifying a functor means defining “where each object and arrow from one category is sent in the other category”. One should note that the composition of two functors is a functor. Let’s consider the example category \( \mathcal{C} \) above.

**Example 2.0.3.** Consider as example the functor \( F \) in the following figure. Every object/arrow of the input category \( \mathcal{C} \) is mapped to an object/arrow in \( \mathcal{D} \), and both identities and compositions are preserved.

---

\(^1\) See Appendix B for a more rigorous definition.
3. DATABASES AS CATEGORIES

In this section, we discuss the basic intuition: there is a tight connection between database schemas and categories, and between database states and functors.

3.1 Schemas are Categories

The schema of a database is naturally represented by a category in which: (i) objects represent schema tables and datatypes, and (ii) arrows represent columns. As an example consider the following relational schema \( S \):

employee(ssn int, name varchar, deptno int);
department(dno int, address varchar);

Where employee.deptno is a foreign key that references department.dno. The categorical representation \( C_S \) for schema \( S \) is shown in Figure 1.

Figure 1: A simple schema as a category

More precisely we state the following:

**Definition 3.1.1. Schemas are Categories:** A database schema \( S \) is a finitely presentable category \( C_S \). Each table \( t \) of the schema \( S \) is represented by a corresponding object \( t \in Ob_{C_S} \), and each column \( c \) in \( S \) with data type \( d \) is represented as \( c: t \to d \). Each table also has an implicit column \( id \) which is represented as an identity map \( id: t \to \text{id} \) of \( t \). Foreign keys are defined as arrows between tables (functions between row identifiers), together with commutative properties for the referenced columns (see Figure 1).

Note that the identity column of each table closely matches the notion of row identifier often used in practical implementations of the relational model. Also note that datatypes are represented uniformly as tables conceptually listing all the values in the domain of an attribute. In this way data columns and foreign key columns are put on equal footing. The definition of a foreign key is constituted by an arrow between the two tables, and the declaration of commutativity of the two paths containing referencing and referenced columns, as shown in Figure 1. The notion of uniqueness of a column, required to model primary keys, requires more powerful mathematics, namely sketches, and its discussion is postponed to Section ??.

3.2 Database states are Set-Valued Functors

Now the question arise: how to represent a database state? A database state is a collection of sets/functions, one for each table/column, that satisfies the schema structure, data types and integrity constraints. In the world of categories it is natural to represent this as a functor from the category \( C_S \) representing the schema to the category Set of all possible sets.

So a database state is a functor in which: each table (object) in the schema is mapped to the set (in Set) of row identifiers stored in that table, and each column (arrow) is mapped to an arrow (in Set) that associates row identifiers to column values for this specific state. This fully characterizes a state, and thanks to the fact that functors preserve identity and composition, it also guarantees that the state is valid or is a model for \( S \). This means that only functors representing valid states can be defined between \( C_S \) and Set.

More formally we say:

**Definition 3.2.1. Database states are set-valued Functors:** A database state of schema \( S \) is a functor \( \gamma: C_S \to \text{Set} \). The functor \( \gamma \) takes each object \( t \in C_S \) to a set \( \gamma(t) \in \text{Set} \) (representing the row identifiers of table \( t \) for this state), and each column \( c: t \to t' \in \text{Arr}_{C_S}(t,t') \) to an arrow \( \gamma(c): \gamma(t) \to \gamma(t') \) in Set that associates each row identifier \( x \in \gamma(t) \) to its corresponding column value in the set \( \gamma(t') \).

One of the strengths of category theory is to allow us to change level of abstraction very naturally. We exploit this here, by representing database states as the objects of a much larger category called \( C_S \)-Set. \( C_S\)-Set thus is the category of all valid database states for schema \( S \). The arrows of \( C_S\)-Set are called natural transformations. Using transformations we can capture database insert and delete operations. In the following section we will use this to provide a semantics for update commands. Various mathematical subtleties related to \( C_S\)-Set will be worked out rigorously in Section ??.

3.3 The Update Category

We are now ready to introduce the notion of update commands in terms of the arrows/transformations in \( C_S\)-Set. We introduce this intuitively here for insert and delete statements, while we postpone to Section ?? a richer construction and rigorous definition. For now we defined a insert/delete command in terms of the arrows in \( C_S\)-Set. In particular, every insert/delete statement is represented by a family of arrows in \( C_S\)-Set, one for each database state (object). An arrow from the objects \( D_1 \) to \( D_2 \) represents the effect of a certain update statement applied to \( D_1 \).

In the following figure we show how an update \( u \) operates on few database states (in light gray you can see other updates).

A richer definition of the semantics of an update command is given in terms of the transformations in \( C_S\)-Set in Section ??, where we will also show that this definition fits nicely with many other constructions representing schema migration, querying etc.

3.4 Queries as Functors

Queries are naturally represented as functors between the category of the schema \( C_S \) and the category representing
the answer schema/targetlist. Let us introduce this idea with a simple example, consider the schema $S$ introduced in Figure 2 and the following SQL query:

```sql
SELECT d.address AS dep_addr
FROM employee e, department d
WHERE e.deptno = d.did AND e.name="carlo";
```

We can represent this categorically as shown in Figure 2. In figure, we show two functors $F$ and $G$ between the schema category $C_S$, an intermediate category $Q_S$ which contains the basic schema plus objects representing the query and the query constants (e.g., $Q$ and $=carlo$ nodes) and arrows connecting these to tables and types relevant to the query, and a simple category containing schema of the query answer $A_S$. The composition properties defined in $Q_S$ corresponds to the condition in the `where` clause of the SQL statement. The two functors are defined very simply, where every objects and arrows with the same names are mapped. The only exception is the value of address in $G$, for which we explicitly state that the path $\text{address} \cdot q\_\text{department}$ in $Q_S$ corresponds to the path $\text{dept\_addr}$ in $A_S$. Note that the directions are.

Very interestingly the definition of query we provide at the schema level is naturally imposed on the corresponding instances $Q_S$.

As we will show more formally in Section 4.1, every select, project, join, union query can be express by chaining at most three functors.

In this section, we showed the basic results of our work and how categories and functors can be used to capture the basic entities in the database world. In the following section, we provide further details on this and, thus, prepare the ground for the more advanced results of Section 4.1. CC: longer and explaining why we need next section.

4. UNDERLYING MATHEMATICS

In Section 3, we gave an explicit definition of the basic idea, that of representing schemas as categories and states as functors. We also gave a brief introduction to updates and queries so that the reader could see how the general theory fit in with these definitions. In this section we explicate these ideas in detail. We will show that the concepts are tight enough that classical theorems in mathematics yield beautiful results, for example a tight link between data migration and queries.

We begin by explaining the setup for migrating data or querying, namely the notion of morphism between schemas, which will simply be a functor. We will then discuss the notion of mapping from one state to another (within the same schema) which will be the basis for our rigorous definition of updates.

At this point we will need to shift gears to describe the data migration functors and queries. While queries are defined solely by schema mappings, they act at an instance level by applying some powerful mathematical machinery. We will conclude this section by discussing data types and calculated fields.

For the reader who is interested in the tight formulations of schema mapping, data migration, queries, updates, and typing, this section will be a careful presentation. The reader who is interested in only the main ideas may skim (or skip) to Section 5 without much trouble.

From here on, we may denote a schema as $C$ rather than $C_S$.

4.1 Morphisms between schemas

A schema can be thought of as a graph in which we can specify when two paths are equivalent. Graph morphisms send objects to objects and arrows to arrows, and a schema morphism does too, with the additional requirement that equivalent paths are sent to equivalent paths; in other words it is a functor. Here is the precise definition.

**Definition 4.1.1.** Let $C$ and $D$ be schemas (categories). A morphism of schemas, denoted $F: C \rightarrow D$, is simply a functor. That is, it is a mapping sending each table $T$ in $C$ to a table $F(T)$ in $D$ and each column $T \rightarrow T'$ in $C$ to a column $F(T) \rightarrow F(T')$ in $D$, such that the composition law enforced in $C$ must also be enforced in $D$.

At this point we have defined schemas and morphisms of schemas, and there is an obvious identity and composition law for morphisms. In other words, we have a category of schemas. We will not really need to use this fact, although it may be convenient for phrasing future results.

**Example 4.1.2.** $DS$: worthwhile? If not, can you come up with something?

There are two possible category structures on the graph

In one, we set $f \cdot g = f \cdot h$ and in the other we do not (so $f \cdot g \neq f \cdot h$). Let's call the first once $C_1$ and the sec-

---

**Figure 2: Representing a query**

![Diagram showing functors F, G, and A with nodes and arrows labeled to represent a database query.](image-url)
ond $C_2$. Considering these as schemas, we will discuss the morphisms between them.

There is a unique functor $C_2 \to C_1$ that sends every object to itself, but there is no functor $C_1 \to C_2$ that sends every object to itself. However, if we drop the requirement that objects are sent to themselves, then there are many functors $C_1 \to C_2$. For example there is one which sends every object in $C_1$ to $X \in \text{Ob}(C_2)$ and every morphism to $\text{id}_X$. As an exercise, the reader may check that there are in fact ten (10) functors $C_1 \to C_2$. Each of these will prescribe a different way to exchange data between $C_1$ and $C_2$.

Currently, in order to create a schema mapping one writes down a set of logical axioms that detail how the data in one schema relates to that of the other schema. Here we suggest that many (if not all) of the mappings which come up in practice can be inferred from functors, as in Definition 4.2.1. Whereas logical axioms can be ad hoc, redundant, and difficult to envision, functors are simple to understand and surprisingly powerful. DS: Can you support this claim?

4.2 A new concept of state transformations

Legend has it that Eilenberg and MacLane spent the effort to invent category theory because they needed to formalize the concept now known as natural transformation. Whereas a functor connects categories, a natural transformation connects functors (we promise not to go on like this!).

The definition of natural transformations between states naturally captures much of the semantics for what remains unchanged when performing an update. Between two set-valued functors $I,J: C \to \text{Set}$, a natural transformation is a way to tie row-ids in instance $I$ to row-ids in instance $J$ such that all facts are preserved on the nose.

With enough uniqueness constraints, this amounts only to inserting rows (and tying rows pre-insert to their corres- lates post-insert) or collapsing duplicate rows into a single one. If data is anonymized (a concept we will discuss in Section 4.1.1) then transformations give a much richer semantics which may have uses in data mining. Either way, the natural “adjointness” machinery which naturally converts axioms which may have uses in data mining. Either way, the natural “adjointness” machinery which naturally converts queries, updates, and pivoting. Moreover, queries can be understood using classical theorems of category theory if given a column $f: c \to c'$ its value in $\delta$ must be sent by $m$ to the value of the $(m_c(r), \delta(f))$-cell in $\delta$. For example, $\delta$ may be the result of inserting some rows into $\gamma$, in which case $m$ is the inclusion of states.

By now we have defined a new category for every schema $C$: the category of states on $C$, which we denote $C \to \text{Set}$. The objects of $C \to \text{Set}$ are states $\gamma: C \to \text{Set}$ and the morphisms are transformations of states. As a trivial example, consider the category with only one object and its identity morphism $[0]:=\lambda$. Then $[0]: \text{Set}$ is the category of sets $-$ a state on $[0]$ is just a set of rows.

For any schema $C$, the category $C \to \text{Set}$ is a topos. That means that there is an internal language and logic for states on $C$ supporting all the usual logical connectives. For example, if $m: \gamma \implies \delta$ is an inclusion of states, the logic of $C \to \text{Set}$ allows us to take the complement of $\gamma$ in $\delta$. However, note that the naive (table by table) complement of one state in another will not generally be a state. The topos logic “takes that into account” and returns the largest possible state contained inside the naive complement.

Example 4.2.2. Consider the category $\mathcal{G} = \bullet \xrightarrow{\ell} \bullet \xrightarrow{x} V$

A state $\delta: \mathcal{G} \to \text{Set}$ is the same thing as a multi-graph; it consists of a set $\delta(V)$ of vertices, a set $\delta(E)$ of edges, and a source and target map between them. A transformation of states $m: \gamma \implies \delta$ is the same thing as a graph morphism; it sends vertices to vertices and edges to edges in such a way that the source and target of each edge is preserved. So for example a graph isomorphism is just an isomorphism in the category $\mathcal{G} \to \text{Set}$.

4.3 Data migration functors

In this section we show that given any morphism of schemas, $F: C \to D$, three data migration functors $\Delta_F, \Sigma_F,$ and $\Pi_F$ are automatically defined. These functors transport data between $C$ and $D$ according to logical rules. However, these rules do not have to be explicitly mentioned; they arise as universal constructions.

In this section we will define these three migration functors. As evidence that these constructions, which seemingly have not been previously discussed in database literature, have “real-world” meaning, we will show that in special cases they naturally compute projects, duplicates, unions, joins, and selects roughly following this dictionary:

<table>
<thead>
<tr>
<th>Migration Functor</th>
<th>SQL command</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta_F$</td>
<td>Project / duplicate</td>
<td>Backward</td>
</tr>
<tr>
<td>$\Sigma_F$</td>
<td>Union / Skolem</td>
<td>Forward</td>
</tr>
<tr>
<td>$\Pi_F$</td>
<td>Join / Select</td>
<td>Forward</td>
</tr>
</tbody>
</table>
These migration functors have been well-known in category theory literature for 50 years, albeit with different intentions. We will explain what is meant in the above table by the “Direction” column shortly.

We should also note that Δ, Σ, and Π can be used in combination to create GLAV views or complex queries. Although we have not defined or even properly described these migration functors yet, we provide an example of a theorem which roughly says that any complex query of this form can be refactored as a project / duplicate, followed by a select / join, followed by a union / skolem.

**THEOREM 4.3.1.** Suppose one has a sequence of functors

\[ C_1 \overset{F}{\longrightarrow} D_1 \overset{G}{\longrightarrow} E_1 \overset{H}{\longrightarrow} C_{n+1}, \]

and consider the migration functor \( X : C_1 \to C_{n+1} \), given by

\[ X = \Delta_{F_1} \Pi_{G_1} \Sigma_{H_1} \Delta_{F_2} \Pi_{G_2} \Sigma_{H_2} \cdots \Pi_{G_N} \Sigma_{H_n}. \]

Then there exists some 3-term sequence

\[ C_1 \overset{F}{\longrightarrow} D \overset{G}{\longrightarrow} E \overset{H}{\longrightarrow} C_{n+1}, \]

and an isomorphism \( X \cong \Delta_F \Pi_G \Sigma_H \) of functors \( C_1 \to C_{n+1} \). Moreover, the process of finding this 3-term sequence is implementable. DS: word choice?

**PROOF.** [?] \( \square \)

DS: Say somewhere that this composition of functors can be thought of as “pipelining.”

Now we will describe the migration functors from a mathematical point of view. The simplest to describe is \( \Delta \) which basically works by projection or duplication.

**4.3.1 The migration functor \( \Delta \)**

We begin with an example. Consider the morphism \( F \) of schemas depicted below; DS: Should \( D \) below have the commutativity relations on it as in (??)?? The idea is the same without them, they add space, we don’t care if the reader thinks they’re there or not. Yet not having them here, and saying that (??) is the running example may add annoying ambiguity.

\[
\begin{align*}
\mathcal{D} &:= \begin{array}{c}
\text{Employee} \\
\text{First} & \text{Last} & \text{String}
\end{array} \\
\mathcal{C} &:= \begin{array}{c}
\text{Employee} \\
\text{First} & \text{Last} & \text{Name}
\end{array}
\end{align*}
\]

Given a database state on \( C \), it should be clear how one would import it to \( D \): one simply drops the Department table entirely, and projects out the Dpt and Mgr column of the Employee table. One can also use \( \Delta \) to duplicate tables or columns. For example, consider the functor

\[
\begin{array}{c}
\mathcal{D} := \\
\begin{array}{ccc}
E_1 & E_2 \\
l_1 & l_2
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\mathcal{C} := \\
\begin{array}{ccc}
E_1 & E_2 \\
d & s
\end{array}
\end{array}
\]

where \( F \) sends \( E_1, E_2 \) to \( E \) and sends \( l_1 \) and \( l_2 \) to \( l \). Given any state \( \gamma : C \to \text{Set} \) on \( C \) the migration functor \( \Delta_F \) duplicates the employee table, as well as projecting out everything but its last name column, and puts that result in \( D \).

In this section we will show that a similar process works for any morphism of schemas. Not only that, but there will be two ways to export a state on \( C \) to a state on \( D \). The distinction between them does not appear to have been discussed before in database literature. DS: how do you feel about that?

Let \( F : C \to D \) be a morphism of schemas. Given a state \( \delta : D \to \text{Set} \), let \( \Delta_F(\delta) : C \to \text{Set} \) denote the composition of functors

\[ C \overset{\delta}{\longrightarrow} D \overset{\Delta_F}{\longrightarrow} \text{Set}. \]

We will call \( \Delta_F \) the re-indexing functor — it is a functor by Lemma 4.3.2 below.

**LEMMA 4.3.2.** Let \( F : C \to D \) be a morphism of schemas. The mapping \( \Delta_F \) described above extends to a functor

\[ \Delta_F : \text{D-Set} \to \text{C-Set}, \]

which in particular means that for any transformation of states \( \delta : D \to \text{Set} \), there is a transformation \( \Delta_F(\delta) : C \to \text{Set} \) of states on \( C \), and the composition law is preserved.

**PROOF.** See [?]. \( \square \)

Given a schema mapping \( F : C \to D \), the functor \( \Delta_F \) imports data from \( D \) to \( C \). We now describe two functors that export data from \( C \) to \( D \). These functors, denoted \( \Sigma_F \) and \( \Pi_F \) are known as the left and right adjoint (respectively) of \( \Delta_F \). The reader can see above that once \( F \) is defined, the migration functor \( \Delta_F \) is not chosen but mathematically given. In the same way, its adjoints \( \Sigma_F \) and \( \Pi_F \) are mathematically given, even if they are more difficult to explain.

We must save the explicit descriptions of \( \Sigma_F \) and \( \Pi_F \) for the appendix, Section ??, where we allow ourselves to assume more category-theoretic background on the part of the reader. However, we can give the following informal descriptions here.

**4.3.2 The migration functor \( \Pi \)**

Let \( F : C \to D \), and let \( \gamma : C \to \text{Set} \) be a state on \( C \); we wish to describe the state \( \Pi_F(\gamma) \) on \( D \). To do so, we will take an arbitrary table in \( D \) and describe the set of rows in the exported state \( \Pi_F(\gamma)(t) \). Given any table \( t \) in \( D \) there are a number of tables \( u \) in \( C \) that are either sent by \( F \) to \( t \) itself, or are sent to some table to which \( t \) has a transitive column. The set \( \Pi_F(\gamma)(t) \) is roughly the join (or “limit”) of all rows in all such \( u \)'s.
**Example 4.3.3.** Recall the functor $F: \mathcal{D} \to \mathcal{C}$ from Diagram (3), and that given a state $\gamma$ on $\mathcal{C}$ the result of $\Delta_F(\gamma)$ is a state on $\mathcal{D}$ that has duplicated the employee table and forgotten everything but the last names. Now suppose we follow $F$ by the functor $\Sigma$, and let $t: \mathcal{E} \to \text{Set}$ be a state on $\mathcal{E}$. One can prove that the result will leave the employee tables alone, and that table $W$ will be their join — in other words, a row in $W$ (more precisely in $\Delta_F(\Sigma_{\mathcal{C}}(\gamma))(W)$) is a pair of employees with the same last name.

**4.3.3 The migration functor $\Sigma$**

Let $H: \mathcal{E} \to \mathcal{C}$, and let $e: \mathcal{E} \to \text{Set}$ be a state on $\mathcal{E}$; we wish to describe the state $\Sigma_H(e)$ on $\mathcal{C}$. To do so, we will take an arbitrary table in $\mathcal{C}$ and describe the set of rows in the exported state $\Sigma_H(e)(t)$. Given any table $t$ in $\mathcal{C}$ there are a number of tables $u$ in $\mathcal{E}$ that are either sent by $H$ to $t$ itself, or are sent to some table with a transitive column to $t$. The set $\Sigma_H(e)(t)$ is roughly the union (or "colimit") of all rows in all such $u$'s.

Note that this process automatically creates skolem variables. For example, consider Diagram (2) and $F: \mathcal{D} \to \mathcal{C}$ respectively to $\mathcal{E}, \mathcal{C}$, and $H: \mathcal{E} \to \mathcal{C}$ so as to conform with the current notation. Given any state on $\mathcal{E}$ (which consists of a set of employees and a first and last name for each), we need to give a state on $\mathcal{C}$. This is done by including a row in the Department table for not only every employee but every employees manager (to the nth power). We would also have to add new rows to the String table as well. These would not be strings, but variables acting like strings, called Skolem variables. These Skolem variables could be manipulated in various ways by state transformations; e.g. we may equate the skolem variable for Alice's manager with the skolem variable for Bob's department's secretary. In Theorem 4.3.4 we will show that, as long as there are only finitely many paths through a both the source and target database schema, there will only be finitely many skolem variables created under the mapping.

**Theorem 4.3.4.** Let $F: \mathcal{C} \to \mathcal{D}$ be a morphism of schemas, and let $\gamma \in \mathcal{C}: \text{Set}$ be a finite state on $\mathcal{C}$. If $\mathcal{C}$ and $\mathcal{D}$ are finite categories then $F(\gamma)$ will be a finite state on $\mathcal{D}$.

**Proof.** For each object $e \in \text{Ob}_\mathcal{D}$, the set of $x$-records of $\Sigma_F(\gamma)$ is given by a colimit of finite sets, whose indexing category is finite. The result is a quotient of (and hence has at most the cardinality of) a finite coproduct of finite sets, which is finite.

**Example 4.3.5.** Let $\mathcal{D} := \text{mgr} \circ E$ with no relations imposed, $\mathcal{D}$ has infinitely many arrows, $\text{mgr}^n$ for all $n \in \mathbb{N}$. Let $\mathcal{C} := \mathcal{E}$ and $F: \mathcal{C} \to \mathcal{D}$ the unique functor. Given a finite set of employees $\gamma: \mathcal{E} \to \text{Set}$, one can load that data into $\mathcal{D}$ by applying $\Sigma_F(\gamma)$. The result will be a state on $\mathcal{D}$ which includes infinitely many skolem variable (one for each choice of employee-record in $\gamma$ and $n \in \mathbb{N}$).

However, if we know that the manager-hierarchy of our business is only 7 deep (i.e. that a 7th-degree manager is his or her own manager) then we can replace $\mathcal{D}$ by the category

and consider the unique functor $G: \mathcal{C} \to \mathcal{E}$. Now Theorem 4.3.4 applies, and we can rest assured that $\Sigma_G(\gamma)$ is a finite state. In fact its cardinality will be precisely $8$ times the cardinality of $\gamma(E)$.

Moreover, if in the future our manager-hierarchy changes (say to allow for 10th degree managers), we can again use migration functors to transform the old data into the new scheme.

**4.4 Typing and calculated fields**

Consider again the category $\mathcal{C}$ from Section ?? Diagram ???. The observant reader may have noticed that a state $\gamma: \mathcal{C} \to \text{Set}$ has no obligation to return the set of strings when applied to the object called $\text{String}$. This predicts an interesting problem: when merging or comparing databases across schemas, there is no reason to expect that the implementation of basic types, such as strings, is the same. In order to enforce that $\gamma(\text{String})$ really is some set of strings (e.g. varchar(40)), one can use the migration functors and an already-established typing system.

This brings us to an interesting point. As mentioned in the introduction, category theory has been remarkably successful in modeling programming languages and advancing the theory of functional programs. For example, monads (functors of a certain kind) have brought notions such as input/output, state, exceptions, and concurrency to the purely functional setting. The goal of PL theory is to reason about programs and prove that they will perform as expected. It is our hope and belief that category theory will help bring that kind of guarantee and reasoning ability to database theory as well.

While it is a bit of a digression, let us take a moment to describe how category theory applies to functional programming languages (such as Haskell and ML). The relevant category $\mathcal{T}$ is that of "types and programs": the objects are simple data types such as Int, String, Bool, as well as more complex data types such as (Int−→Int) whose values are themselves functions. The arrows in $\mathcal{T}$ are programs, for example the program that takes a string and returns its length as an integer; this is an arrow String−→Int. For each object in $\mathcal{T}$, there is an intended "set of values" $V(t)$ and each program $t \to t'$ is a function from the values of $t$ to the values of $t'$. In other words, $V: \mathcal{T} \to \text{Set}$ is a functor.

We call the category $\mathcal{T}$, together with the values functor $V$...
a typing system, meaning a category of types and programs for which every object and arrow is fully implementable on a computer. Note the similarity between $T$ and a database schema $C$: in both cases the objects are types and the arrows are some way to take values of one type and produce values of another; in both cases the values of those types are given by a set-valued functor. The only difference is that the types in $C$ are most often “user-defined” and their values are changing, whereas the types in $T$ seem closer to “god-given” and their values are permanent.

**Definition 4.4.1.** Let $(T, V : T \rightarrow \text{Set})$ be a typing system as above and let $C$ be a category. We call a functor $G : T \rightarrow C$ a $(T, V)$-typing of $C$. An object in $T$ (or its image under $G$) will be called a typed object and a morphism in $T$ (or its image under $G$) will be called a calculated field.

A $(T, V)$-typed state on $C$ consists of a state $\gamma \in C \cdot \text{Set}$ and a transformation $\gamma \rightarrow \Pi \in V$ of states on $C$, or equivalently a transformation $\Delta \in V$ of states on $T$.

Note that under this definition any computer program $P$ can serve as a foreign key column connecting its input type $X$ to its output type $Y$; we merely write it as an arrow $X \xrightarrow{f} Y$ in $T$. We did not define morphisms of $(T, V)$-typed states in Definition 4.4.1 for fear of overwhelming readers. It can be found in Definition 3.6.4.

**Example 4.4.2.** Suppose each employee of a company has a salary and a travel allowance, and we wish to calculate the sum of these two fields. The relevant schema is

\[
C := \begin{array}{c}
\text{salary} \\
\text{travel}
\end{array}
\]

\[
\begin{array}{c}
\xymatrix{ & \ast \ar[r]^{\pi_1} & \mathbb{R} \ar[d]^f \ar[r] & \ast \\
\ast \ar[ru]_f & & & \ast \ar[u]^\pi_2 \ar[r] & \mathbb{R} \ar[r] & \ast \ar[ru]_+}
\end{array}
\]

The composite $f; + : \ast \rightarrow \mathbb{R}$ is a calculated field. However, nothing about this schema forces that the elements of $\mathbb{R}$ or $\mathbb{R} \times \mathbb{R}$ are anything like real numbers or pairs of such, nor that the column $+$ really produces the sum. To enforce these constraints, we use a typing as above.

Let $T$ be the category

\[
T := \begin{array}{c}
\ast \ar[r]^{\pi_1} & \mathbb{R} \ar[r] & \ast \\
\ast \ar[r]_+ & \ast \ar[r]^{\pi_2} & \ast \ar[r] & \mathbb{R} \ar[r] & \ast
\end{array}
\]

and let $V : T \rightarrow \text{Set}$ be appropriate functor $\text{DS: Carlo: help}$. Under Definition 4.4.1, every object in $C$, except for $E$, is a typed object, and both projections $\pi_1, \pi_2$ and the sum of real numbers $+$ are calculated fields, whereas salary and travel are not.

There is an obvious functor $G : T \rightarrow C$, so we have a typing of $C$. A $(T, V)$-typed state $\delta$ on $C$ consists of a set $\delta(x)$ for every object $x$ in $C$, together with an assignment of a value to each cell in $T$. In other words, it is what we want.

The point here is that the way in which categories are used in PL theory fits in perfectly with the way we propose to use categories in DB theory. We can exploit that connection to make a bridge between these two fields of research. In Section 6.1 we discuss this issue a bit more to integrate a few advanced results.

### 4.5 The update category, revisited

In Section 3.3.

We start from this rather loose definition of an update command that we can formally state as follows:

**Definition 4.5.1.** Let $C$ be a schema. An update command on $C$ is a function $U : \text{Ob}_C \cdot \text{Set} \rightarrow \text{Ob}_C \cdot \text{Set}$; the subcategory of $C \cdot \text{Set}$ on which $U$ is defined is called the domain of definition of $U$. Let $\text{UpCmd}_C$ denote the monoid of update commands on $C$. The multiplication law for this monoid is simply sequential application of update commands. Two update commands are considered equal if they are the same as functions.

This simple definition already allows us to define equivalence for chains of updates statements. However, we are interested in giving a tighter definition based on

We are now interested in the category of updates, as opposed to the category (or monoid) of update commands. We want each object in this category to be a state and each morphism $A \rightarrow B$ to be an update command applied to $A$ which returns $B$. We say that two updates $f, g : A \rightarrow B$ are equivalent if commands which generated $f$ and $g$ are equal, i.e. if these two commands are identical when applied to any state. It will turn out (Proposition ??) that this definition is not arbitrary from a categorical perspective, but instead "suggested" in the sense that it comes from a major tool of category theory called the Grothendieck construction.

**Definition 4.5.2.** Let $C$ be a schema. The category of updates on $C$, denoted $\text{UpCmd}_C$, has as objects the set $\text{Ob}_C \cdot \text{Set}$ II \{fail\} of states on $C$ plus a fail state. If $\gamma$ and $\delta$ are states, then a morphism $f : \gamma \rightarrow \delta$ is an update command $U \in \text{UpCmd}_C$ such that $U(\gamma) = \delta$, and a morphism $\gamma \rightarrow \text{fail}$ is any update command which fails on $\gamma$. For the composition law, we say $f_1 \ast f_2 = f_3$ if the underlying update commands also satisfy that equation, $U_1 \ast U_2 = U_3$, i.e. if this composition law would hold for any input state.

**Proposition 4.5.3.** Let $C$ be a category and let $\text{UpCmd}_C$ be the monoid of update commands, considered as a category with one object. There is a functor $F : \text{UpCmd}_C \rightarrow \text{Set}$ where the unique object of $\text{UpCmd}$ is sent to the set $\text{Ob}_C \cdot \text{Set}$ II \{fail\}, and each update command is sent to the corresponding function $\text{Ob}_C \cdot \text{Set} \rightarrow \text{Ob}_C \cdot \text{Set}$. The Grothendieck construction of $F$ is precisely the category $\text{UpCmd}_C$ defined in Definition 4.5.2.

**Proof.** Obvious to those who know and can work with the definition of Grothendieck constructions.

\[
\square
\]

**Remark 4.5.4.** In Definition 4.3.1 we defined an update command to be a function from states to states. Surprisingly, it seems that updates are often functorial; i.e. they
respect the notion of “transformation of states” defined in Definition 4.2.1. The precise structure of update commands, such as those found in SQLs data manipulation language, should be studied using category theory to determine

- whether all update commands are indeed partial functors \( U : C \rightarrow \text{Set} \)

- whether there is a nice characterization of standard updates within the category of all partial functors \( C \rightarrow \text{Set} \),

- and whether, if there are some partial functors that are not standardly considered updates, such functors may have new capabilities that have not been explored.

In Section ?? we will discuss transactions and triggers and show that category theory can provide a useful semantics for these.

5. CATEGORIES FOR CLASSICAL RESULTS

In the previous sections, we provided the user with a natural representation of databases in terms of categories and functors, what remains to be shown is that this specific application of category theory to databases is constructive, and that is a valid alternative to prove useful results. We attempt this in this section by showing how we can use the framework introduced above to prove some of the classical results in database theory.

CC: - schema mapping as a functor - extensions of aggregates and UDF

6. MORE ADVANCED RESULTS

In the previous sections, we showed that category theory naturally extend set theory, and, thus, naturally allows to represent database problems, and its rigorous foundations allows us to prove some of the classical theorems from the database literature. This section is devoted to suggest how using these conceptual tools we can imagine to tackle more advanced problems.

6.1 Integrations with PL

We described in Section 4.1 how category theory is typically applied in the theory programming languages, and how this meshes with our application to databases. In Section 6.0 we will also mention Grothendieck constructions are useful [?] and how they relate to our work. Harper’s work also discussed “Kinds” a category \( K \) together with a map \( K \rightarrow \text{Cat} \). Here, each kind has a category of states, rather than a mere set of states, and these are the types of that kind. Many of the above ideas on set-valued functors can be recast to category-valued functors and again integrate well with Harper’s theory. One could also use functors valued in topological spaces or vector spaces and perhaps find use in spatial or temporal databases.

As a quick example of the link, one of the big successes in programming language theory was the use of monads to describe effects such as those encountered in IO, exceptions, concurrency, etc. A monad on a category \( C \) is a functor \( M : C \rightarrow C \) with some properties. The simplest such example is the list monad which takes a type and produces its list type. In database theory the same monad might be useful to relax the notion of “state transformation” \( m : \gamma \rightarrow \delta \) (from Section 4.2), and instead allow each row in \( \gamma \) to map to a sequence of rows in \( \delta \).

6.2 Grouping

In this section we show that grouping a table by a few of its columns is functorial. That is, it fits in perfectly with the rest of the theory of this paper.

Let \( C \) be a schema; one can picture a 5-column table in \( C \) as a functor \( F : H \rightarrow C \) where \( H \) is the category to the right

There is a functor \( F : G \rightarrow C \) where \( T_1, T_2 \) are sent to \( T \) and labels are otherwise preserved which will allow us to group by columns 4 and 5. Hopefully, the way one does this for an arbitrary subset of the columns is clear.

At this point we have functors

\[
G \xrightarrow{G} H \xrightarrow{F} C
\]

so, as in Section 4.3.1 we can take any database state \( \delta \) on \( C \) and apply \( \Delta_D \) if we want to isolate table \( T \), or we can apply \( \Delta_F \) to that to get a state \( \gamma := \Delta_{DS} \gamma(\delta) \) in \( G \rightarrow \text{Set} \).

It should be clear how this would look: two tables and a foreign key column pointing from the first to the second, where we intend to group by the columns of the second. In other words, for every element in \( \gamma(T_2) \), we want a table with data columns \( C_1, C_2, C_3, \) i.e. a state on

We have the following proposition.

**Proposition 6.2.1.** There is a functor

\[
G \rightarrow \text{Set} \rightarrow X \rightarrow \text{Set} \rightarrow \text{Ob}_X\rightarrow \text{Set}
\]

where \( X \) is the schema

and \( X \) is typed by the set of \( G \)-tables, \( \text{Ob}_{X} \). DS: Carlo: help

**Proof.** ***

Proposition 6.2.1 says that given a choice of how to group some table \( T \), the grouping operation is functorial.

**Remark 6.2.2.** The same idea works for indexing. Given a field \( c : X \rightarrow T \) in a schema \( C \), we can index table \( X \) by type \( T \). Suppose that \( T \) has value set \( V \rightarrow T \). Form the fiber
6.4 Transactions

A transaction is a series of updates which must be performed as a unit, or at least act as though it is performed as a unit. In this section we describe the relationship between transactions and updates in the language of category theory.

The main point is to relate transactions to updates; each will be a morphism in an appropriate category and we want a functor between them. The set of objects in these two categories will be the same, namely the set of states on a given schema \(C\). The difference is in the morphisms; the morphisms in the category of updates are all sequences of updates, whereas the morphisms in the category of transactions are all sequences of transactions. Clearly, every transaction yields a sequence of updates, but some sequences of updates are not transactions.

**Definition 6.4.1.** Let \(C\) be a schema and let \(\text{Up}_C\) be the category of updates on \(C\). A category of transactions on \(C\) is a subcategory \(\mathcal{X}\) of \(\text{Up}_C\) that has the same set of objects, \(\text{Ob}_X = \text{Ob}_{\text{Up}_C} = \text{Ob}_C\). We denote the inclusion functor by \(L: \mathcal{X} \rightarrow \text{Up}_C\).

Note that under the above definition there may be many different categories of transactions on \(C\). That is, one can decide which updates (or sequences of updates) will count as transactions and which will not.

The beauty of this approach is that the updates in a sequence of transactions can be done in any order, so long as the resulting path is equivalent to the “default” path. Let us make this more precise. To say that \(t_1\) and \(t_2\) are transactions with underlying update sequences \(r_1, r_2, \ldots, r_n\) and \(s_1, s_2, \ldots, s_n\) respectively, is to say that \(L(t_1) = r_1 * r_2 * \cdots * r_n\) and \(L(t_2) = s_1 * s_2 * \cdots * s_n\). And of course, \(L(t_1 * t_2) = r_1 * \cdots * s_n\) accordingly. However, there may be more than one sequence of updates that compose to equal \(L(t_1 * t_2)\). In a category, two different sequences of arrows may be declared equal, and we have declared to sequences of updates to be equal if they have the same effect. Thus the transaction scheduler could choose any sequence of updates whose composition is equal to \(L(t_1 * t_2)\).

Recall that we defined updates using a notion of update commands. Similarly we could define transaction commands, a monoid which acts on the set \(\text{Ob}_C\) of states, just as \(\text{UpCmd}_C\) does. A result analogous to Proposition 4.5.3 is easily proved. In fact, it would be interesting to study the category of submonoids of \(\text{UpCmd}_C\), to classify various notions of transaction on a given schema.

**Remark 6.4.2.** The above does not explore transactions in all their glory. There are promising connections with the “state monad” in PL theory that will be useful to explore in future work. This monad would allow one to model programs which have the capacity to access the state of the database, use it in a computation, and then update the state. Monads are specifically designed to allow one to keep track of the history of such transactions.

6.5 Triggers

A database schema is supposed to give a certain semantic to the data. Triggers are designed to uphold this semantic; given an update which would violate a constraint, triggers are employed to rectify the situation. In this paper, we do not describe triggers which act outside the database (like sending a supervisor an email whenever a certain event occurs), we only describe those that act on the state of the database; we call these corrective triggers. We can describe corrective triggers categorically as follows.

\[
\begin{array}{ccc}
X \times T & V \rightarrow & X \\
\downarrow & \downarrow & \downarrow \\
\downarrow & \downarrow & \downarrow \\
V & T & \mathcal{G}
\end{array}
\]

and, after removing the arrow \(c\) (using a migration functor), apply the above operation to group by \(V\). The result is a map \(V \rightarrow \text{Ob}_X\), that sends every value to the set of \(X\) records that have that value in the \(c\) field.

6.3 Aggregates

Let \(C\) be a schema and \(S = \text{Ob}_C\) its set of states. Suppose given a function \(h: S \rightarrow T\), where \(T\) is any set. For example \(T\) could be the set of integers and \(h\) could send any state to its total number of rows or the sum of cells in a given column of \(C\). Then aggregation simply becomes a combination of grouping, as defined in Section 6.2 and the function \(h\), which could be user-defined.

**Example 6.3.1.** A scientist is measuring the spring constant for springs made of different materials. He or she wants a table of materials vs. spring constants. We will consider the map of schemas

\[
\mathcal{G}':=\begin{array}{ccc}
\text{sample} & \xrightarrow{F} & \text{experiment} \\
\text{mtrl} & \xrightarrow{r} & \text{mtrl}
\end{array}
\]

A single experiment can be viewed in \(\mathcal{G}'\), whereas the totality is viewed on \(\mathcal{G}\). Let \(S = \text{Ob}_{\mathcal{G}'}\) be the set of states on \(\mathcal{G}'\). A data set for one spring can either be seen formally as an element of \(S\) or visually as a scatterplot of force-extension data for that spring.

By Hook’s law, the force vs. extension curve of a spring is assumed linear, of the form \(y = kx + b\). During the data reduction process the scientist will use some program \(h: S \rightarrow \mathbb{R}\) which assigns to each data set a best-fit value of the spring constant \(k\). By grouping, we can functorially transform a state on \(\mathcal{G}\) into a state on the schema

\[
\begin{array}{ccc}
\text{experiment} & \xrightarrow{S} & \text{mtrl} \\
\text{mtrl} & \xrightarrow{R} & \mathbb{R}
\end{array}
\]

which sends each experiment to its data set. We can then use migration functors and typing to add a calculated field given by \(h: S \rightarrow \mathbb{R}\) to yield a state on the schema

\[
\begin{array}{ccc}
\text{experiment} & \xrightarrow{S} & \text{mtrl} \\
\text{mtrl} & \xrightarrow{R} & \mathbb{R}
\end{array}
\]

This will be a table of materials and their spring constants.
Suppose that $C$ is a sketch (which includes information like “table $T$ is the product of table $A$ and table $B$”) and $B$ is the underlying category (with only foreign key type constraints). Then any state on $C$ will restrict to a state on $B$, whereas a state on $B$ must be “corrected” in order to give a state on $C$. In other words, there are functors $F: C \to \text{Set}$ and $G: \text{B-Set} \to C\text{-Set}$ such that $FG = \text{id}_{C\text{-Set}}$, where $G$ is the functor that takes a state on $B$ and formulaically adds rows to table $T$ to ensure that it becomes the product of $A$ and $B$. Updates to a $C$-set $\delta$ will generally land in $\text{B-Set}$ because the updater either does not know about the constraint or wants it to be taken care of automatically. The trigger is basically $G$ — it is employed to take the resulting $\text{B-set}$ and “rectify it” to give a new $C$-set. Whereas this is generally done with a piece of code, the semantics should be considered as a functor $G$ as above.

6.6 RDF and semi-structured data

Suppose that $C$ is a schema and $\delta: C \to \text{Set}$ is a state. A beautiful construction of category theory, called the Grothendieck construction, allows one to convert the functor $\delta$ into a new category $\text{El}(\delta)$, called the category of elements of $\delta$, which is surprisingly similar to an RDF graph. The objects of $\text{El}(\delta)$, like vertices in an RDF graph, correspond to the entities in $\delta$, i.e., one for each row in each table. The arrows of $\text{El}(\delta)$, like edges in an RDF graph, correspond to the attributes of the entities in $\delta$, i.e., an arrow points from an entity to its value in some cell. So the subject and value in a (subject, predicate, value) triple correspond to two objects in $\text{El}(\delta)$, and the predicate corresponds to an arrow between them.

In fact, there is a canonical functor $\pi: \text{El}(\delta) \to C$, where $\text{El}(\delta)$ acts as the RDF triple store (or RDF graph), $C$ acts as the RDF schema, and $\pi$ sends each entity and predicate to its type as declared in $C$. Functors such as $\pi$ can encode semi-structured data as well. In other words, such a functor (or RDF triple store) may not come from a relational database state on $C$, for example because such a state would have null values or multiple values in a single cell, or may not conform to the commutativity constraints declared in $C$. Thus the Grothendieck construction sends database states into a more “relaxed” world of semi-structured data in a functorial way.

6.7 Pivoting

DS: If we want to be sure of no nulls in the pivoted thing, we have to enforce that $Ce$ is a fiber product as drawn below in $\mathcal{P}$. But if we allow nulls in the pivoted thing then we don’t need to get into it (we can stay in categories, rather than needing sketches). If we allow nulls then we aren’t in the world of database states, but “semi-structured data” as described in Section 6.6. What do you think?

We can pivot any state on a schema of the following form:

$\mathcal{P} := \begin{array}{c}
\delta \\
\downarrow C_o \\
\downarrow \downarrow D_T \\
T \\
\end{array}$

where the labels of the objects stand for: row, cell, domain, table, column, data type — respectively. One can change the number of columns in a table or the data types of those columns by changing the state on $\mathcal{P}$. Pivoting turns the state on $\mathcal{P}$ into a new schema whose tables are the elements of $T$, columns are elements of $Co$ with data types given by $t$, etc.

In this section we define pivoting only for schemas in which the tables have no foreign keys, just data columns. Allowing foreign keys can be done along the same lines $\text{DS: should we put it in an appendix}?$.

What is amazing, from a categorical perspective, about pivoting is that it is functorial — in other words it naturally respects the categorical structure we defined in Section 6.2 and works hand-in-hand with two of the migration functors $\Delta$ and $\Sigma$.

A state on only the bottom row of $\mathcal{P}$ can be transformed into a bipartite graph with tables on one side, data types on the other, and an arrow from each table to its data type. Such a graph is in fact a category. Moreover, morphisms of states on $\mathcal{P}$ are sent by this construction to functors between categories. That is, we have a functor

$\text{Piv}_0: \mathcal{P} \to \text{Set}$

(proof we prove this as part of Proposition 6.7.2 which works only at the schema level. We need the following definition in order to see the full power of pivoting.

**Definition 6.7.1.** Let $\text{Data}$ denote the category for which an object is a pair $(C, \delta: C \to \text{Set})$ consisting of a schema and a state on it; and for which a morphism $a: (C, \delta) \to (C', \delta')$ consists of a pair $(a, a')$ where $a: C \to C'$ is a map of schemas and $a': \delta \to a'\delta'$ is a transformation of states.

In terms of RDF graphs, the above is easier to understand: it’s just a commutative square of categories

$$
\begin{array}{c}
\text{Gr}(\delta) \overset{a}{\longrightarrow} \text{Gr}(\delta') \\
\downarrow \downarrow \downarrow \\
C \overset{a}{\longrightarrow} C'
\end{array}
$$

**Proposition 6.7.2.** There is a functor $\text{Piv}: \mathcal{P} \to \text{Data}$, sending a state $\gamma$ on $\mathcal{P}$ to an object $(C, \delta)$ where the set of tables in $C$ is $\gamma(T)$, the set of all columns is $\gamma(Co)$, each column is sourced in a table by $\gamma(s)$ and has values in a data type by $\gamma(t)$, the set of all rows is given by $\gamma(R)$, the set of cells is given by $\gamma(\text{Ce})$, etc.

**Proof.** If $F$ is the schema morphism

$\begin{array}{c}
T' := \\
\downarrow \downarrow \\
\gamma(T') \\
\downarrow \downarrow \\
\gamma(Co) \\
\end{array}$

the migration functor $\Sigma_F$ sends states on $\mathcal{P}$ to states on $\mathcal{Q}$. A state on $\mathcal{Q}$ is a graph, and there is a functor sending graphs to categories. Combining this with the $\Delta$ migration functor applied to the obvious inclusion $\mathcal{P} \to \mathcal{Q}$ gives us $\text{Piv}_0$ as above.

Applying this same construction to the top row of $\mathcal{P}$ yields another such functor, written $\text{Piv}_1: \mathcal{P} \to \text{Cat}$, and a natural transformation $m: \text{Piv}_1 \to \text{Piv}_0$. The requirement on $\mathcal{P}$-sets $\delta$ that $\delta(Ce) = \delta(R) \times \delta(Co)$ enforces that the functor $\text{Piv}_1(\delta) \to \text{Piv}_0(\delta)$ is a discrete op-fibration, which means that $\text{Piv}_1(\delta) = \text{Gr}(\gamma)$ for some state $\gamma: \text{Piv}_0(\delta) \to \text{Set}$. We can consider $m$ as a functor $m: \mathcal{P} \to \text{Cat}$, and the preceding sentence implies that it serves as a functor $m: \mathcal{P} \to \text{Data}$. If we take $\text{Piv} := m$, it is clear that it has the prescribed properties.
Remark 6.7.3. It is interesting to notice that applying
the Grothendieck construction to a state on $P$ is similar to
"reification" in RDF triples stores and schemas. The arrows
(cells) become new objects which point to both their subject
and value. This is just another case of how natural concepts
in category theory almost magically recreate ideas that may
have seemed ad hoc or complicated without it.

CC: (mention many prove one)
- transactions
- schema mapping extensions... beyond weak acyclicity
- PL integration... etc..

7. RELATED WORK

DS: The following papers discuss sketches as models for
database schemas:

- Zinovy Diskin. “Databases as diagram algebras: spec-
ifying queries and views via the graph-based logic of
sketches.” Research Report FIS/LDBD-96-02. Frame
Inform System, Riga, Latvia, 1996

- Zinovy Diskin, Boris Kadish. “Variable set seman-
tics for keyed generalized sketches: formal semantics
for object identity and abstract syntax for conceptual
modeling” Data & Knowledge Engineering. Volume
47 Issue 1, October 2003

- Michael Johnson, Robert Rosebrugh, and R. J. Wood.
“Entity-relationship-attribute designs and sketches.”

I claim that we are doing something more here.

- Many of these papers were either not clear or not writ-
ten for a computer scientist.

- These papers are too flexible: so much of the theory
can be done with just categories (not sketches) that
you give up a lot of power by jumping so far. For exam-
ple “data migration” does not really exist for sketches
like it does for categories.

- These papers do not discuss calculated fields.

- These papers do not discuss the transformations of
states that make so much of our machinery work.

- These papers do not discuss updates, grouping, trigg-
ers, transactions, pivoting, RDF, etc.

- The conception of aggregates in some of them is sur-
prisingly unlike mine.

- These papers do not show any examples in table form,
thus making it difficult to understand.

- These papers do not emphasize commutative diagrams,
and never suggest anything like $mgr^7 = mgr^8$.

8. CONCLUSIONS

category can be very useful for DB

APPENDIX

A. CATEGORY THEORY