Exploring Category-Theoretic Approaches to Databases

Aaron J. Walter
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
walte774@morris.umn.edu

ABSTRACT

As organizations are formed, split up, merge, or collaborate, they must often work to integrate different database structures. Database migration accounts for a substantial portion of information technology (IT) budgets, and database migration failure is common. This paper explores how to avoid database migration failure. Concepts from category theory, a branch of mathematics, can help.

Keywords

Relational database, database migration, category theory

1. INTRODUCTION

Database migration accounts for approximately 40% of IT budgets, and it is believed that over half of database migrations fail [1]. This motivates a framework for databases that allows us to guarantee correctness of translations between database schemas. Category theory serves as the basis of this framework, which is primarily concerned with relational databases. Category theory is a branch of mathematics which focuses on the relationships between objects and the structures of those relationships. In Section 2 I provide the mathematical and computer science background of the paper – relations, relational databases, graphs, and the building blocks of category theory. In Section 3 I describe the category-theoretic formulation of relational databases proposed by Spivak and Wisnesky. In Section 4 I draw conclusions about the practicality and applicability of this framework.

2. BACKGROUND

I will give background on relations, which are the mathematical foundation for relational databases. I will describe graphs as a foundation for categories. I will discuss the basic elements of category theory – categories, functors, natural transformations, and adjoints.

2.1 Relations

Definition 1. An *n-tuple* is an ordered list of n elements.

The ordered pair $(0, 0)$ is a tuple with $n = 2$.

UMM CSci Senior Seminar Conference, October 2020 Morris, MN.

An n-tuple is a generalization of an ordered pair – n elements given in order from left to right, enclosed in parentheses.

Definition 2. Given sets $X_1, X_2, ..., X_n$, the *Cartesian product* $X_1 \times X_2 \times ... \times X_n$ is the set of all n-tuples $(x_1, x_2, ..., x_n)$ such that each x_i is in X_i.

If X is the set $\{apple, ham\}$ and Y is the set $\{pie, burger\}$, the Cartesian product $X \times Y$ is the set $\{(apple, pie), (apple, burger), (ham, pie), (ham, burger)\}$.

Definition 3. A *relation* over sets $X_1, X_2, ..., X_n$ is a subset of the Cartesian product $X_1 \times X_2 \times ... \times X_n$.

The set $\{(apple, pie), (ham, burger)\}$ is a relation over X, Y.

More often than describing a relation by explicitly giving its members, we will describe a relation by giving the rule or property that its members satisfy. For example, we might define a relation R to be the following set:

$$R := \{(x, y) \in \mathbb{Z} \times \mathbb{Z} : x < y\}$$

Here, $\mathbb{Z}$ is the set of integers: $\{..., -2, -1, 0, 1, 2, ...\}$. The Cartesian product $\mathbb{Z} \times \mathbb{Z}$ is the set of all ordered pairs of integers, and the relation R is the set of all of those ordered pairs such that the first integer in the pair is less than the second. So the ordered pair $(1, 2)$ belongs in R, but the ordered pairs $(2, 1)$ and $(1, 1)$ do not. If the ordered pair (a, b) is in R, we might write this as aRb, and say "a is related to b (by R)". It can be helpful to think of a relation as having a short verb phrase. For R, that phrase might be "is less than".

It's important to recognize that relations can be defined on the iterated Cartesian product of a single set with itself (for example, $\mathbb{Z} \times \mathbb{Z}$ above) or on the Cartesian product of several distinct sets. For example, we might define a relation S on the set $X \times Y \times Z$, where X is the set of students who have attended the University of Minnesota Morris, Y is the set of professors who have taught a course at Morris, and Z is the set of semesters where Morris has held courses. We could define S to be the set of all 3-tuples (x, y, z) – each with one student x, one professor y, and one semester z – such that x took a course with y in z. For example, if the author took a course with David Roberts in the spring of 2020, the 3-tuple (Aaron Walter, David Roberts, spring 2020) would belong in S. We call a relation n-ary if its objects are n-tuples. For example, R above is a binary (2-ary) relation, while S is a ternary (3-ary) relation.

Relations often have useful properties, such as symmetry, transitivity, and reflexivity. A relation that has all of these properties is called an *equivalence relation*. A reader who is unfamiliar with these concepts can refer to an undergraduate textbook on set theory, such as [5].

2.2 Relational databases

The *relational model of databases* was proposed by Codd in 1970 and later refined by Darwen and Date in 1995 [3]. It provides "a simple and intuitive method for defining a database, storing and updating data in it, and submitting queries of arbitrary complexity to it" [3]. A *query* is a structured request for information. If we built a relational database of information about the University of Minnesota, Morris, it might have a record (Abstract Algebra, David Roberts, spring 2020). This record is a tuple. A record belongs to a collection of similar records, called a table. For example, this table might also include the records (CSci Senior Seminar, Elena Machkasova, fall 2020) and (Algorithms and Computability, Peter Dolan, fall 2020). If all the records in this collection can be read as "Course x was taught by y in z", we could think of this collection as representing an analogous relation. So in relational databases, tables (collections of records) are relations, records (rows in a table) are tuples, and attributes (the columns in a table) are the underlying sets of the relation that table represents. Every attribute in a table is associated to a particular table (when its entries are foreign keys) or to a particular type of raw data.

An ideal relational database adheres to four key principles [3]:

1. Every row/column intersection has exactly one value. In terms of relations, we might say "every record is an n-tuple", where n is the number of columns in the table.
2. The order of rows in a table doesn't matter. This reflects the fact that a relation is a set, and the order of elements in a set doesn't matter.
3. The order of columns in a table doesn't matter, as long as each column is associated with the correct attribute name.
4. Rows in a table are unique. This reflects the fact that duplicate elements in a set don't matter.

A *database schema* is a description of the structure of a database – its tables, the attributes of those tables (other tables or data types) and any "business logic" that goes with them. As an example of business logic, a schema might specify that a worker's next pay check amount is equal to their pay rate times the amount of hours they've worked over the last pay period. A schema is a blueprint for a database instance.

An *instance* of a database schema is a presentation of data that obeys the structure of that schema.

2.3 Structured Query Language

Structured Query Language (SQL) is a programming language used to manage database systems. While SQL has historically been used mostly in the context of relational database management systems (RDBMS), its use has expanded in recent years, for example to databases designed for processing "big data" [7]. SQL has three core components: a database definition language (DDL) which allows for the specification of database schemas, a data manipulation language (DML) which allows for the access and modification of stored data, and a data control language (DCL) which allows for the configuration of security protocols for a database [7]. The database definition language functions essentially the same way as we will see later in the category-theoretic framework. The data control language is outside the scope of the question of database migration. So, for our purposes we will focus on the data manipulation language, which has features which violate the four principles from Section 2.2.

Darwen and Date condemn SQL as a "perversion" of the relational model of data [4]. They note that SQL has certain properties which are forbidden by the relational model. For example, SQL-managed relational databases allow for duplicate rows in a table and allow null values in columns rather than requiring every entry correspond to the attribute name of its column. Further, SQL can operate on relations in a "tuple-at-a-time" level rather than handling them as a set. These properties can be problematic for data migration. Graph theory will serve as our foundation for a category-theoretic treatment of relational databases, as a potential alternative to SQL.

2.4 Graph theory

Definition 4. A *graph*

$$G := \{V, A, s, t\}$$

has a set of vertices V, a set of arrows A, and functions $s, t : A \to V$ which take arrows in A to vertices in V. s can be thought of as sending each arrow to its source vertex, and t can be thought of sending each arrow to its target vertex.

Readers familiar with graphs might recognize that Definition 4 describes a *directed multigraph*. This paper is not concerned with other types of graphs, so the distinction is ignored here. Further, note that what we call arrows are often called edges by graph theorists. We choose the term arrow to emphasize the idea of movement, or transition between states. Arrow is also the term that Spivak and Wisnesky use [9].

Table 1 describes a graph G.

Table 1: An example graph G

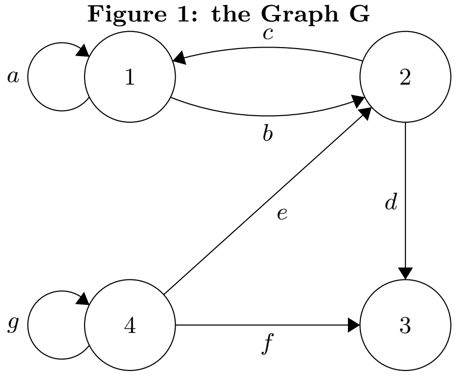
V
1
2
3
4

A	s	t
a	1	1
b	1	2
c	2	1
d	2	3
e	4	2
f	4	3
g	4	4

Here, G is a graph with four vertices (1 through 4) and seven arrows (a through g). The entries in the s and t columns describe the source and target, respectively, of the arrow in the corresponding row. For example, arrow c starts at vertex 2 and goes to vertex 1.

Definition 5. Given a graph G, a *path* in G is a sequence of arrows a_1,a_2,...,a_n in G such that the target of each arrow in the sequence is the source of the next arrow in the

Figure 1: the Graph G



sequence. We say that the *length* of the path is the number of arrows composed this way. We also allow the trivial path of length 0 on each vertex, denoted id_v for a vertex v. Here, id stands for identity.

2.5 Category theory fundamentals

The mathematical field of category theory is concerned with relationships between structures. The subsections below define important terms. Definitions are taken or adapted from [6].

2.5.1 Categories

Below we give the formal definition of a category. The intuition behind the definition is this: a category is a graph (see Definition 4) with an additional bit of expressive power: we can declare any two paths to be equivalent. This power will allow us to inject "business logic" directly into a description of a database. By "business logic" we mean rules that ensure the integrity of our data. For example, we might want to guarantee that, in a workplace database, a manager works in the same department as the employees they manage. When we represent a category with a diagram, the *objects* of that category are the vertices, and the *morphisms* are the arrows. A morphism is a structure-preserving map between objects in a category. We will see that morphsisms are the key to understanding path equivalences categorically.

Definition 6. A *category* C has

1. A collection of *objects* called $\mathbf{Ob}$(C).
2. A set of *morphisms* for every pair of objects c,d called $C(c,d)$. We call $C(c,d)$ a *hom-set*.
3. An *identity morphism* $\mathbf{id}_c : c \to c$ for every object $c \in \mathbf{Ob}(C)$.
4. For every three objects $c, d, e \in \mathbf{Ob}(C)$ and two morphisms $f \in C(c, d)$, $g \in C(d, e)$, a morphism $f; g \in C(c, e)$ called the *composite of f and g*.

Note: we normally write function compositions with the outermost function on the left, and the innermost on the right. We use a different convention to denote composition of morphisms: the left morphism is the first arrow in the path and the right morphism is the last. This motivates the use of ; as our symbol to denote morphism composition rather than $\bigcirc$.

We will sometimes write $c \in \mathbf{Ob}(C)$ as simply $c \in C$, and $f \in C(c, d)$ as $f : c \to d$. When we do this, we say that c is the *source of f* and that d is the *target of f.*

The components of C must obey the following conditions:

a. *unitality*: For any morphism $f : c \to d$, composing with the identities at c or d does not change the resulting morphism: $\mathbf{id}_c; f = f; \mathbf{id}_d = f$.

b. *associativity*: For any three morphisms $f : c_0 \to c_1$, $g : c_1 \to c_2$, $h : c_2 \to c_3$, the following are equal: $(f; g); h = f; (g; h)$. We simply write this as $f; g; h$.

The objects of a category C can be anything we can represent with a mathematical construct. Morphisms are one-way connections between objects. Importantly, every object is connected to itself (for an object c, that self-connecting morphism is $\mathbf{id}_c$). An object can be connected to many other objects, or just itself. Condition 4 of the definition ensures that when two objects are connected indirectly, we name a connection between them by composing the morphisms that create that indirect connection.

Composing morphisms in a cateogry can be thought of as concatenating arrows in a graph. If we want to compose morphisms f and g, the 'target' of f must match the 'source' of g, and we get $f; g$.

For example, let's say for some category C we have $\mathbf{Ob}(C) = \{a, b, c\}$. So C has three objects. We know that each has an identity morphism. Let's say that there are also four morphisms $e : a \to b$, $f : b \to c$, $g : c \to b$ and $h : c \to c$. Because of the unitality and associativity conditions, certain composite morphisms must exist. $e; f : a \to c$ is one of them.

Definition 7. We say a morphism $f : A \to B$ is an *isomorphism* if there exists a morphism $g : B \to A$ such that $f; g = \mathbf{id}_A$ and $g; f = \mathbf{id}_B$. We say that f and g are *inverses*, and that A and B are *isomorphic.*

In category, we will sometimes say a category is *small.* The distinction is not important for our purposes – we will deal with categories described by diagrams, which will always have a finite number of vertices. Hence, all of our schemas and instances we represent as categories will be small.

Definition 8. Later on we will make use of these special categories:

- A *discrete category* has only the identity morphisms as its morphisms. Graphically, a discrete category is a collection of vertices with no nontrivial paths.
- Given a category C, its *opposite category* denoted C^{op} is C with all of its morphisms reversed. Graphically, it is the graph of C with the directions of all the arrows switched.
- **Set**, the category of sets, has sets as its objects and functions between sets as its morphisms. For example, for sets A and B, $\mathbf{Set}(A, B)$ is the set of all functions from A to B.
- **Cat**, the category of small categories, has small categories as its objects and functors as its morphisms. For example, for categories C and D, $\mathbf{Cat}(C, D)$ is the set of all functors from C to D.

2.5.2 Functors

Definition 9. Let C, D be categories. A *functor* $F : C \to D$ is a map of objects in C to objects in D and morphisms in C to morphisms in D such that

1. For every object $c \in \mathbf{Ob}(C)$, we specify an object $F(c) \in \mathbf{Ob}(D)$.
2. For every morphism $f : c_1 \to c_2$ in C, we specify a morphism $F(f) : F(c_1) \to F(c_2)$ in D.

These constituents must satisfy two properties:

a. For every object $c \in \mathbf{Ob}(C)$, we have $F(\mathbf{id}_c) = \mathbf{id}_{F(c)}$.

b. For every three objects $c_1, c_2, c_3 \in \mathbf{Ob}(C)$ and two morphisms $f \in C(c_1, c_2)$, $g \in C(c_2, c_3)$, the equation $F(f; g) = F(f); F(g)$ holds in D.

Condition a says that a functor sends each object c's identity morphism to the identity morphism in D that belongs to the same object that F sends c to.

Condition b says that when composing morphisms, it doesn't matter whether we do so before or after sending them through F. This can be thought of as being a structure-preserving property.

2.5.3 Natural transformations

Definition 10. Given two categories C and D, and two functors $F, G : C \to D$, to specify a *natural transformation* $\alpha : F \Rightarrow G$,

- For each object $c \in C$, we specify a morphism $\alpha_c : F(c) \to G(c)$ in D called the *c-component of* α.
- These components must satisfy the *naturality condition*, which states that for every morphism $f : c \to d$ in C, the equation $F(f); \alpha_d = \alpha_c; G(f)$.

Natural transformations can be thought of as being higher-order objects in category theory. While functors are a way of translating from one category to another, natural transformations are a way of translating from one functor to another.

We use functors and natural transformations to determine when two categories are equivalent.

Definition 11. Given categories C, D, we say *C and D are equivalent* if there is a functor $L : C \to D$ and a functor $R : D \to C$ such that $L; R$ and $R; L$ are both natural transformations.

2.5.4 Adjoints

Definition 12. Given C, D categories and $L : C \to D$, $R : D \to C$ functors, we say L *is left adjoint to* R (and R *is right adjoint to* L if for any $c \in C$ and $d \in D$, there is an isomorphism of hom-sets $\alpha_{c,d} : C(c, R(d)) \xrightarrow{\cong} D(L(c), d)$ that is *natural in c and d.* In the database context, our hom-sets will simply be sets, so all that is required for isomorphism is that those sets are the same size.

Given a morphism $f : c \to R(d)$ in C, its image $g := \alpha_{c,d}(f)$ is called its *mate.* Similarly, the mate of $g : L(c) \to d$ is f.

To be natural in c and d, our isomorphism $\alpha_{c,d}$ must be such that for all morphisms $f : c' \to c$ in C and $g : d \to d'$ in D, going from $C(c, R(d)$ to $D(L(c'), d'$ by the path $\alpha_{c,d}; D(L(f), g)$ is the same as taking the path $C(f, R(g); \alpha_{c',d'}$.

3. CATEGORY THEORY IN DATABASES

When applying category theory to databases, there are two key aspects of databases that we would like to be able to represent: database schemas and instances on database schemas. Both of these can be treated as categories. When thinking of a schema as a category, its objects (vertices) are the schema's tables, and its morphisms (arrows) are the schema's columns [8].

Because relational databases are based on the mathematics of relations, it makes sense to define path equivalences as a special kind of equivalence relation.

Definition 13. Given a graph G and the set $Path_G$ of paths in G, a *categorical path equivalence relation* is an equivalence relation $\simeq$ on $Path_G$ that has the following properties:

- Given paths p, q, If $p \simeq q$, then $\mathrm{src}(p) = \mathrm{src}(q)$ and $\mathrm{tgt}(p) = \mathrm{tgt}(q)$. Recall that src and tgt are the source and target functions of G.
- Given paths $p, q : a \to b$ and arrows $m : z \to a$ and $n : b \to c$, $mp \simeq mq$ and $pn \simeq qn$.

The first property says that our equivalence relation must respect the sources and targets of paths. The second property says that when we have equivalent paths, applying some arrow to both paths results in paths that are equivalent.

Definition 14. A *categorical schema* C is a pair $C := (G, \simeq)$ of a graph G with a categorical path equivalence relation $\simeq$ on G.

Definition 15. Given a categorical schema $C := (G, \simeq)$ on a graph $G{:=}(V,A,s,t)$ an *instance on* C, denoted I, has the following:

- For every vertex $v \in V$, a set $I(v)$.
- For every arrow $a : v_1 \to v_2 \in A$, a function $I(a) : I(v_1) \to I(v_2)$.
- For every path equivalence $p \simeq q$, the equality $I(p) = I(q)$ holds.

Because of how we've defined instances, we can see that an instance on C is a functor from C to **Set**.

Definition 16. Given categorical schemas $C := (G, \simeq_C)$ on a graph $G{:=}(V_G,A_G,s_G,t_G)$ and $D := (H, \simeq_D)$ on a graph $H{:=}(V_H,A_H,s_H,t_H)$, a *translation F from C to D*, written $F : C \to D$ consists of

- a function $V_F : V_G \to V_H$ which maps vertices in C's graph to vertices in D's graph.
- a function $A_F : A_G \to Path_H$ which maps arrows in C's graph to paths in D's graph.

We require that A_F preserves sources, targets, and path equivalences.

Definition 17. **Sch** is the category with categorical schemas as its objects and translations between categorical schemas as its morphisms.

For every database schema S, there is a category $S - \mathbf{Inst}$ of instances on that schema [8]. Its objects are instances on S (which are functors from S to **Set**) and its morphsisms are natural transformations between those instances.

3.1 Functorial data migration

We've seen that database schemas are essentially categories, and we've seen that instances on schemas and translations between schemas are essentially functors. Now we need a way to convert an instance on one schema to an instance on another schema. This is the problem of data migration, and we solve it with the following special functors.

Definition 18. Given a translation $F : S \to T$ from a schema S to a schema T and an instance I on T, the following *data migration functors* are induced:

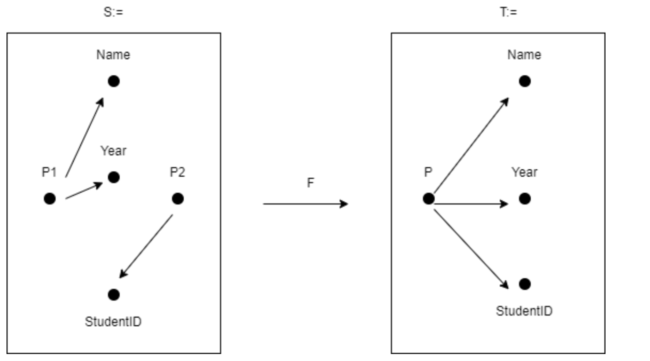
- $\Delta_F : T{-}\textbf{Inst} \to S{-}\textbf{Inst}$ is defined to be $\Delta_F(I) = F; I$
- $\Sigma_F : S - \textbf{Inst} \to T - \textbf{Inst}$ is the left adjoint of Δ_F.
- $\Pi_F : S - \textbf{Inst} \to T - \textbf{Inst}$ is the right adjoint of Δ_F.

We call Δ_F the "pullback" data migration functor. It takes instances of T and returns instances of S. If we have an instance of T, applying Δ_F to it recovers that instance in the structure of S.

We call Σ_F and Π_F the "push-forward" data migration functors, although they push S-instances to T-instances in different ways. In some sense, Σ_F unites tables and Π_F multiplies tables. How exactly Σ_F and Π_F behave depends on the source and target schema and the choice of translation F.

We provide a concrete example that illustrates schemas, instances, a translation between schemas, and the application of the data migration functors.

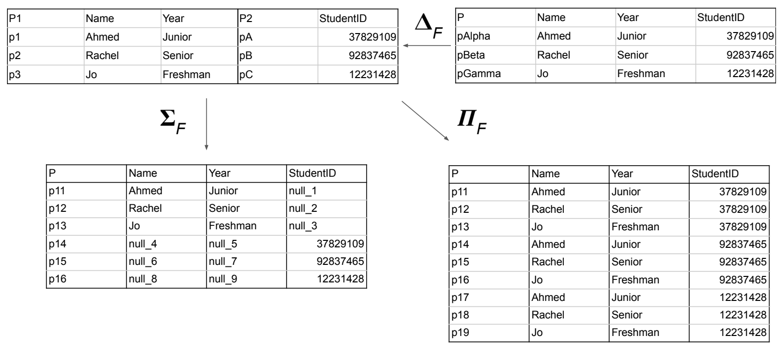
Figure 2: A translation F between schemas S and T



In Figure 3.1, the translation F maps the tables P1 and P2 in S to the table P2 in T. It maps the tables Name, Year, and StudentID in S to their analogues in T. With this mapping of objects, there is no choice of how to map our arrows. The arrow from P1 to Name in S must go to the arrow from P to Name in T, because our translation must respect arrow sources and targets. The other arrows are mapped in an analogous way.

In Figure 3.1, the T-instance I is in the top right corner of the figure. The pullback Δ_F splits the table P into the tables P1 and P2 we would expect to have in an S-instance. The push-forward functors Σ_F and Π_F turn this new S-instance back into T-instances, but in different ways. We can see how Π_F behaves like multiplication: the records in the new P are all the possible combinations of records from the tables P1 and P2 in the pullback. We can see how Σ_F unites tables: the records in the new T-instance are all of the records from the tables P1 and P2 put into P, with unknown information left null.

Figure 3: Applying the data migration functors


Δ_F
Σ_F
Π_F

P1	Name	Year	P2	StudentID
p1	Ahmed	Junior	pA	37829109
p2	Rachel	Senior	pB	92837465
p3	Jo	Freshman	pC	12231428

P	Name	Year	StudentID
pAlpha	Ahmed	Junior	37829109
pBeta	Rachel	Senior	92837465
pGamma	Jo	Freshman	12231428

P	Name	Year	StudentID
p11	Ahmed	Junior	null_1
p12	Rachel	Senior	null_2
p13	Jo	Freshman	null_3
p14	null_4	null_5	37829109
p15	null_6	null_7	92837465
p16	null_8	null_9	12231428

P	Name	Year	StudentID
p11	Ahmed	Junior	37829109
p12	Rachel	Senior	37829109
p13	Jo	Freshman	37829109
p14	Ahmed	Junior	92837465
p15	Rachel	Senior	92837465
p16	Jo	Freshman	92837465
p17	Ahmed	Junior	12231428
p18	Rachel	Senior	12231428
p19	Jo	Freshman	12231428

We've seen three distinct but closely related data migrations. The definitions from Section 2 guided us in our construction of an appropriate translation F and the adjoints of Δ_F. The structure-preserving properties of these constructs guarantee data migration that obeys the specifications of the source and target schema.

Now consider the possible consequences of data migration gone wrong. For example, if a university tries to reformat how it stores records of which courses students have taken and the grades students got in those courses, incorrect migration might result in mismatched foreign keys (e.g. the new instance claims a student has taken a course they haven't actually taken). Such an outcome may result in chaos if the university has not backed up its old database beforehand. In contexts like healthcare and engineering, faulty data migration might result in loss of human life and resources.

3.2 Databases with controlled vocabularies as categories

In one of Spivak's foundational works on CT in databases, he considers only databases where variable values come from controlled vocabularies [8]. A controlled vocabulary restricts possibilities for the value of a variable. For example, a database where the strings for the Name column in the table Employees must come from the list "Abdul, Rachel, William" is using a controlled vocabulary for that field. This approach is restrictive, but we will discuss how it's expanded on in 3.3. Spivak shows that the category **Sch** of database schemas is equivalent to **Cat**, the category of small categories [8]. We've been thinking of categories as graphs with path equivalences, so we can see why **Sch** and **Cat** are equivalent: objects in categories are vertices in schemas, morphisms in categories are paths in schemas, path equivalences in categories are path equivalence relations in schemas, and functors between categories are translations between schemas.

This means that every theorem about small categories becomes a theorem about databases using controlled vocabularies. Spivak uses graphs to represent database schemas, like we've seen in Figure 3.1. Graphs are commonly used to represent databases, but Spivak treats these schemas in a category-theoretic way. The additional bit of power that CT gives to these representations is the ability to declare rules about path equivalences. These path equivalences usually have a clear translation to "business logic" in a real-life application. For example, in a university database, we might declare that an undergraduate student's academic advisor must be part of the faculty for that student's major. We can represent this rule as a path equivalence: Student.major = Student.advisor.discipline.

3.3 CT in databases with data types

Spivak and Wisnesky build on the work discussed in 3.2 to expand the kinds of databases we can consider categorically [9]. They extend the data migration functors to behave appropriately with schemas with data types. They use a construct called a *typing*, the specifics of which we will not go into here. The important thing to note about a typing is that it uses a natural transformation to ensure that records' typed entries have the appropriate data type after migration.

3.4 Categorical data integration

In later work, Brown, Spivak, and Wisnesky address data integration with category theory [2]. Data integration is the process of unifying database schemas and instances. For example, if two hospitals with different database systems are united by a merger, bringing their database systems together would be an exercise in data integration. It is beyond the scope of this paper to discuss the mathematical background behind this technique, but it serves as an example of the wider practical applicability of CT to databases.

3.5 Functorial query language

Functorial query language (FQL) is a programming language that allows for the creation of queries to a relational database built using the category-theoretic framework. Almost any relational query can be written in the form $\Delta_F \Pi_G \Sigma_H$ for some functors F, G, H [8]. Categorical Query Language (CQL) is the successor project to FQL. We note the existence of these languages here as evidence that the category-theoretic approach to databases works in real-life contexts, but we will not discuss their implementation.

4. CONCLUSIONS

The category-theoretic approach to relational databases has potential as a replacement for SQL in RDBMS (in the form of FQL or its successor CQL) and as a pedagogical tool – both to introduce category theory to programmers and computer science students, and to introduce relational database principles to mathematicians. Representing categories as graphs with path equivalences is straightforward, and allows for the construction of functors, natural transformations, and common database operations just by drawing pictures. More importantly, the structure-preserving properties of our category-theoretic objects help ensure correct data migration. Since faulty data migration can result in mismatched keys, raw data of incorrect typing, and instances that violate business logic, correct data migration is essential to protecting human life and resources.

Acknowledgments

This paper would not have been possible without the time and effort of others. In no particular order, I'd like to thank Elena Machkasova for providing feedback and facilitating our senior seminar course meetings, Peter Dolan for advising me on this project, Nic McPhee for being my second faculty reviewer, my alumni reviewer Nathan Beneke for their feedback, the other students in the course for providing feedback and sharing their own scholarship, and my supportive friends and family.

5. REFERENCES

[1] P. A. Bernstein and L. M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, Sept. 2008.

[2] K. Brown, D. I. Spivak, and R. Wisnesky. Categorical data integration for computational science. *CoRR*, abs/1903.10579, 2019.

[3] H. Darwen. *Relational Database*, page 1519–1524. John Wiley and Sons Ltd., GBR, 2003.

[4] H. Darwen and C. J. Date. The third manifesto. *SIGMOD Rec.*, 24(1):39–49, Mar. 1995.

[5] H. B. Enderton. *Elements of set theory.* Academic press, 1977.

[6] B. Fong and D. I. Spivak. *An invitation to applied category theory: seven sketches in compositionality.* Cambridge University Press, 2019.

[7] Y. N. Silva, I. Almeida, and M. Queiroz. Sql: From traditional databases to big data. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, page 413–418, New York, NY, USA, 2016. Association for Computing Machinery.

[8] D. I. Spivak. Functorial data migration. *Information and Computation*, 217:31 – 51, 2012.

[9] D. I. Spivak and R. Wisnesky. Relational foundations for functorial data migration. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 21–28, New York, NY, USA, 2015. Association for Computing Machinery.