Introduction to databases

Introduction to database schemas

CONTENT

SHARE ON 

Introduction

One of the primary advantages of databases over other, more simple data storage options is their ability to store information in an orderly, easily queryable structure. These features are derived from the fact that databases implement *schemas* to describe the data they store.

A database schema serves as a blueprint for the shape and format of data within a database. For relational databases, this includes describing categories of data and their connections through tables, primary keys, data types, indexes, and other objects. With NoSQL schemas, this often involves organizing data according to the most important anticipated query patterns.

In either case, understanding the value of your database's schema and how best to design and optimize it for your needs is crucial. This guide will focus on what database schemas

are, the different types of schema you might encounter, why they're important, and what to keep in mind when designing your own schemas.

Why are database schemas important?

Database schemas are important for many reasons.

Your data will almost always include some regularity to it, regardless of its source or application. Some data is highly *regular*, meaning that it all can be described by the same patterns. Some data is much more *irregular*, but even so, its *metadata*, contextual data about the data itself, will often still be regular.

Database schemas tell the database what your data is and how to work with it. Database schemas help the database engine understand these patterns which allows it to enforce constraints on the data, respond with the right information when queried, and manipulate it in ways that users request.

Good schemas tend to reduce implicit information in favor of making it visible to the system and its users. Schemas in relational databases can reduce information redundancy, ensure data consistency, and provide the scaffolding and structures needed to access and join related data. Within non-relational contexts, good schemas enable high performance and scalability by aligning the storage format with the access patterns that are essential to your application.

Defining physical vs logical schemas

Before we further, we should introduce a few definitions. Two terms that are potentially confusing are *physical schema* and *logical schema*. These two terms can convey different meanings depending on the *context* in which they are used.

For the purpose of this article, we are mainly talking about logical and physical schemas when *designing database schemas*.

When designing database schemas

When talking about designing database schemas, a **logical schema** is a general design for organizing data into different categories, defining properties of the data, and determining the best structure for database items. This general document has no implementation details and is therefore platform-agnostic. It can be taken as a blueprint and implemented in a variety of database systems.

In this same context, a **physical schema** is recognized as being the next step in the design process where implementation-specific details are worked out. The names of different entities, constraints, keys, indexes, and other items are identified and mapped onto the logical schema. This provides a specific plan for implementation using a given database platform.

In this context, logical and physical schemas are different stages of a design process. The goal of the process is to iteratively develop an implementation plan from a set of requirements by first laying out the abstract qualities of the data and then later mapping that organization to the tool set and language of a database system you want to use.

When discussing database architecture

The other context where physical and logical schema are sometimes seen in regards to databases is in the physical and virtual architecture of the actual database software.

In this context, the **logical schema** refers to the visible database entities that users interact with. This means objects like tables, keys, views, and indexes are abstractions that users create and manipulate using the database software. The layout of these items within the system are part of the logical schema that the database presents.

In this same context, the **physical schema** refers to the way that the database software handles the data, files, and storage when interacting with the filesystem. For example, the physical schema of the database architecture can determine whether the system stores a separate file for each database or each table and determines how those can be partitioned across multiple servers.

Static vs dynamic schemas

Another important categorization that can help clarify the differences between schema in relational and non-relational databases is the difference between static and dynamic

schemas.

Static schemas are the type of schemas generally associated with relational databases. They are defined ahead of time as a definition of the shape that data must follow to be accepted by the system. The database system has the ability to enforce these patterns when using static schema because static schema is an assertion of the desired state that the database system can validate input against.

In contrast, **dynamic schemas** are much more prevalent in non-relational contexts. Dynamic schemas are less rigid and might lack *any* preconceived organizational structure. Instead, dynamic schemas *emerge* based on the qualities of the data that is entered into the system. While many non-relational databases can store information with an arbitrary internal structure, regular patterns tend to emerge with most real world use cases.

Because dynamic schemas are emergent structures, the database system cannot use them as a conformance tool. However, they are still incredibly important to understand and develop around as a user. Understanding what your data will look like in a general sense and how your applications will need to interact with it will help you choose structures that fulfill your requirements, perform well, and avoid unnecessary inconsistency.

Designing database schemas

Now that you understand some of the different types of database schemas, how do you go about designing one for your project? Designing effective schemas takes thought and practice, as well as a thorough understanding of the problem domain and the systems that will use the data.

The design process looks quite different depending on the type of database you are designing the schema for. Specifically, the design process for static schemas differs from that of dynamic schemas. Practically speaking, these end up aligning to differences between designing for relational databases (static) and non-relational databases (dynamic).

General tips

Although there are differences between schema design for relational and non-relational databases, there are some *general* tips that are applicable with any schema development.

Since many of these are important to the beginning of the design process, it makes sense to discuss these first.

Learn about your data

One of the first steps in designing schemas should always be to learn about your data and domain. It is impossible to develop a good database design without understanding the information it will manage and context in which it will be used.

While you will likely not know all of the features of your data in the beginning, learning as much as you can about the data that your system is expected to manage is essential for design.

Some questions you should try to answer include:

- Broadly speaking, what will the data be?
- Which attributes are important to record?
- How large will your total dataset be?
- How rapidly will the system accumulate new data?
- Will your data be highly regular?

Understand usage patterns

Similarly, designing a database schema without understanding user requirements is as problematic as it is with other software design. If you are not an expert in the domain in which the data will be used, you need to consult someone who is to guide you on the requirements.

You should ask yourself questions like:

- Are the most common queries predictable?
- How many concurrent users or clients will there be?
- How much data will be touched by typical operations and queries?
- Will the majority of requests be read queries or write queries?
- What data will be queried together regularly?
- Do most operations target individual records or aggregate many records?

Develop a naming convention

While it might not seem important, designing a naming convention and following it rigorously will help during both development and regular usage.

Naming and styling conventions help minimize the amount of mental work you need to perform when naming new entities. Similarly, conventions allow users to safely assume a pattern when accessing different items within your schemas. Some database systems or types of databases already have popular naming conventions, which you can follow to avoid surprises and avoid the need to develop your own standards.

Some style and naming conventions you might want to consider:

- How should you use upper and lowercase lettering for systems that are case-sensitive?
- When should items use the plural of a word versus the singular?
- Should multi-word names separate words with underscores, dashes, or other delimiters?
- Should full names always be used or are abbreviations permissible in some cases?

Designing schemas for relational databases

Relational databases are often considered flexible, general purpose solutions. Their ability to process ad-hoc queries allows the same database to serve different applications and use cases. Because of this, when designing schemas for relational databases, your end goal is usually to represent your data in a way that promotes flexibility while minimizing the opportunity for data inconsistencies to enter the system.

Developing a logical schema

Relational schema designs often start with a logical schema, as discussed in a previous section.

You map out the data items you want to manage, their relationships, and any attributes important to consider without regard to implementation details or performance criteria. This step is important because it collects all of your data items in one place and allows you to sort through the way they relate to one another on an abstract level.

You can begin sketching out tables that represent specific data items and their attributes. This mapping process is often best represented by entity-relationship (or ER) models. *ER*

models are diagrams that visually represent data objects by defining item types and their attributes and then connecting these to map out relationships and dependencies.

ER models are frequently used in early stage schema designs because they are very good at helping you figure out what distinct entities you have, what attributes must be managed, which entities are related to one another, and the specific nature of their relationship. Using ER model diagrams to represent your logical schema gives you a solid plan for *what* you want your database design to be without commenting on implementation-specific details.

Developing a physical schema

Once you have a logical schema, your next step is to figure out specific implementation details by creating a physical schema (as discussed in a previous section). The physical schema will determine exactly how you want to commit your plan using the database structures and features available to you.

The first step is often to go through each of your database entities and determine your primary key field. The primary key is used to uniquely identify each record within a table as well to bind records together from different tables. When a relationship exists between two entities in the logical schema, you will have to connect the two tables in the physical schema by referencing the primary key in one table as a foreign key in the other. The direction of this relationship will impact the performance and ease in which you can join different entities together when using your database.

Another consideration you will want to think through during this stage are the predicted query patterns. Certain tables and fields within these tables will be accessed much more frequently than others. These "hot spots" are good candidates for database indexes. Database indexes significantly speed up retrieval of commonly accessed items at the cost of worse performance during data updates. Determining which columns to index initially will help you balance these concerns and define the most critical places for indexes in your system.

Normalizing your data structures

During this process, you might find that it's easier to extract certain elements from logical entities into their own independent tables. For instance, you may wish to extract shipping address from a customer so that multiple shipping addresses can be associated with a single customer and so that product orders can reference a specific address. These changes can be thought of as part of a process is called normalization.

Database normalization is a process that ensures that your database represents each piece of data once and doesn't allow updates that would result in inconsistencies. Normalization is a huge topic that, for the most part, is outside of the scope of this guide, but you part of the physical schema design process involves figuring out the level of normalization to seek and transforming data entities as necessary to achieve that goal.

Designing schemas for non-relational and NoSQL databases

The design process for non-relational databases often looks quite different. A large part of this difference stems from the fact that often, non-relational databases are chosen to allow for high performance on a limited number of predefined queries.

Determining your primary queries

Non-relational database schemas are often designed in tandem with the application that will use them. The schema reflects the specific needs of the application and, in a sense, is a custom structure designed to fit the mold developed by the application.

Because of this close relationship, it is important to determine what queries your database must be optimized to respond to. The first step is figuring out what queries your database will need to run. Since you don't have a data structure yet, these will be pseudo queries, but understanding what data your application will need to perform certain operations is your first objective.

Once you have a good idea of what queries your application will need to perform, you need to select the most important ones to focus on. These are the queries that your application performs often and cannot afford to wait for.

Defining which of your queries are the most important tells you the exact access pattern your data structure needs to optimize around. The way that the database system stores and represents data will have a huge impact on its ability to quickly retrieve and manipulate data items.

Design your initial schema around your primary queries

Now that you know your most essential access patterns, you can start to develop a schema to match these queries.

Your first step in this process will be to determine the exact information required to be returned by each query. Then, map out what it would look like to store all of the

information to respond to a query in a single entity.

For instance, if your application will be querying your database to retrieve user profile information, your starting point should likely be to assume that all of the users profile information can be stored in a single place.

Combine and deduplicate data entities where possible

After you've determined the attributes that are needed and mapped out what it would look like to store all items related to each query in a single entity, check for overlaps. The idea is to consolidate data entities where possible to reduce the number of separate items your system will maintain. The greater number of distinct entity types you maintain, the greater chance for inconsistency and update performance problems to arise.

Some of these overlaps will be fairly obvious. Cases where one query returns a subset of the attributes that another query does can be safely collapsed into a single entity.

Other times, it may be more difficult to determine how to map the information for your queries. Non-relational databases are often not great at coalescing data from multiple entities in a single query, something relational databases excel at through joins. So when certain attributes or entities are present in multiple queries, you may have to make a choice in how best to represent that data.

Determine where your application can fill in the gaps

For some queries, your application may need to do part of the work of assembling data instead of relying on the database to respond with all of the relevant information in a single query. For example, if you need to handle customer information and their associated orders, it might make sense to store orders in a different category and reference them by ID in your customer objects.

Some database systems cannot easily join this information by following the references between your objects. Instead, your application may need to query the customer first and then make additional queries for each of the related orders using the order IDs you discovered.

Performing these operations in your application code can help work around the limitations of some non-relational databases. This is often a better option than attempting to maintain a great deal of information within a single entry or attempting to duplicate data many times for many different types of database objects. Those options could result in very poor performance and data consistency.

That being said, it will be important to test and tune your application code and database schemas once they are both online to ensure that you are not trading good application performance for fast database operations. A good rule of thumb is to use the database's capabilities whenever possible since they are highly optimized for information retrieval and manipulation and to supplement within your application as required.

Determine appropriate partition keys

For highly scalable, non-relational databases, users often have to determine a partition or sharding key. These keys will be used to split datasets among various servers to improve performance and responsiveness.

Finding the right partition keys is highly dependent on your data and your workloads. Some general rules, however, can help guide you.

It is best to try to choose a partition key that has a fairly regular distribution of keys. For instance, if you need to distribute customer data, their birth month would typically lead to a decent distribution. In contrast, if you are selling winter clothing, sign up month would not be a good partition key since your products' seasonality would likely affect the distribution of keys. Applying a hashing algorithm to your candidate data can also sometimes help to distribute your key space more evenly.

Another consideration is whether your workloads are read or write heavy. If you have a read-heavy application, you likely want to choose a partition key that will allow you to write as much related data to a single server as possible. This will help you avoid having to read from many servers each time you need to retrieve related data.

On the other hand, if you have write-heavy workloads, it is often preferable to spread the writes over as many servers as possible. If each request ends up writing data to the same server, you will not gain much performance for write-intensive operations.

Wrapping up

Designing effective database schemas takes patience, practice, and often a lot of trial and error.

To start, you have to try to develop a good idea of what your data will look like, how your applications will use it, and what usability and data integrity requirements are required.

Afterwards, your goal is to develop a schema that reflects your data's specific features and facilitates the type of use cases you anticipate.

Schema design, like any other type of design, is an iterative process. Expect to change your design as your understanding of the problem space deepens and as real world performance data becomes available. While you may have to evolve your schema over time, starting off with a solid foundation will both aid you in this process and reduce the likelihood of dramatic, disruptive schema changes in the future.

RELATED ON PRISMA.IO

Prisma defines the characteristics of its data in the **data model** section of the **Prisma schema** file. Check out the linked documentation to learn more about how these concepts apply to Prisma.

You might also want to take a look at the **Prisma schema API reference page** to get an overview of how to use the various features.

Prisma is an open-source database toolkit for Typescript and Node.js that aims to make app developers more productive and confident when working with databases.

About the Author(s)

Justin Ellingwood

Justin has been writing about databases, Linux, infrastructure, and developer tools since 2013. He currently lives in Berlin with his wife and two rabbits. He doesn't usually have to write in the third person, which is a relief for all parties involved.

Previous
← Comparing database types: how database types evolved to meet different needs

Next
Intro (don't panic) →

Edit this page on GitHub

PRISMA'S DATA GUIDE

A growing library of articles focused on making databases more approachable.

Made with ❤️ by Prisma