The Pragmatic Programmers

Facets of Ruby

Series editor: Noel Rappin

Rails Scales!

Practical Techniques for Performance and Growth

Cristian Planas

edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

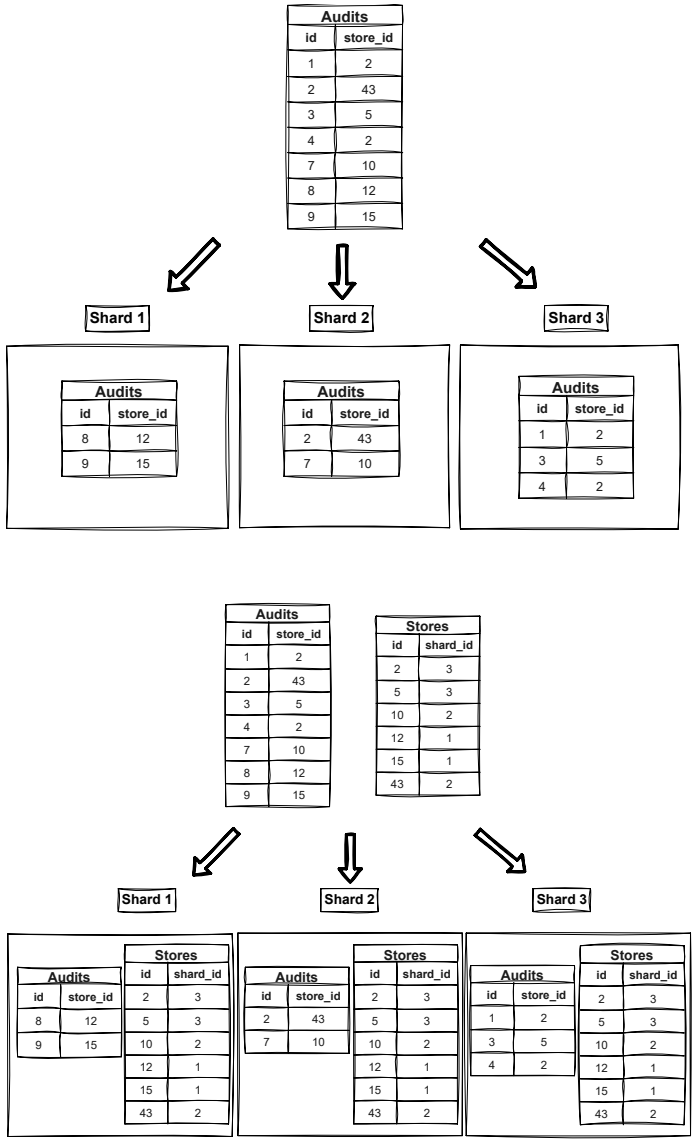
Introducing Horizontal Sharding

If this is the first time that you have read about sharding, you may be wondering how it works in practice. What is the exact strategy used to define the data held by each one of those shards? How can you implement sharding without utterly breaking the functionality of your application?

There is a very common scenario in modern web applications, particularly in B2B services: datasets in which the whole data "hangs" on the account. For example, when a company opens an account in a cloud-based SaaS to manage its human resources, it doesn't expect its data to interact in any way with the data of other accounts. In fact, it would create a huge issue if there was any kind of data leak between accounts! In products like the one we are commenting on—an HR management service—the vast majority of the data can be easily partitioned with no feature loss. In this case, it would be acceptable from a product perspective—even somehow ideal—if each account had its own totally isolated database.

Let's take our movie business as an example. Fortunately, the data model you have in your hands is perfect for applying some sharding. The key you want to use to generate the shards is `store_id`. Still, so far, none of the features in the application seem to require the usage of sharding…until now. Our new requirement has arrived—we are going to introduce auditing in our application. Every time something relevant to the store occurs, we will create an object and store it. This is how it could look in the figure on page 4.

What you have just seen described is an ideal sharding scenario. It is so ideal that it rarely happens in reality, even in use cases in which data sharding is commonly used. The thing is, even if all the data introduced by the customer can be completely sharded, there will be data that is shared. The most common example is that all instances of the application will probably need to connect to a complete `accounts` table that at least allows them to know the shard assigned to a given account, but there are many others. Fortunately, this kind of "global" data tends not to suffer volume issues in the same way that customer-associated data does, and therefore you can replicate it across all shards without causing scalability problems.



After this explanation, you may think that sharding is an obvious choice, almost something that you should implement preemptively in all your applications. That's not the case. The reality of the tech world is not the massive scale that we are discussing here; it's significantly smaller volumes of data. Moreover, sharding notably increases complexity, and complexity is the silent killer of tech businesses. Maybe that's why Rails didn't support sharding out of the box until fairly recently.

Vertical Sharding

Yes, the term "horizontal sharding" implies the existence of "vertical sharding." While vertical sharding is less useful for scalability purposes, it's still a technique that deserves to be explained.

The difference between horizontal and vertical sharding is the following: as we have seen, in horizontal sharding, we "break" the table by creating multiple tables with the same schema as the original one and dividing the rows between them. In other words, the rows are divided, but the columns stay the same. Vertical sharding is the other way around: we break the original table into multiple tables that have a part of the original schema while maintaining all the rows in all the tables. This can make sense, particularly in tables with many columns.

For example, take the films' table. In our application, it's still quite small. It only has five columns. However, imagine the following situation: in the future, we have implemented multilanguage capability, introducing two columns per language in the films' table: one holding the title (en_title', es_title, fr_title and so on) and another holding the plot (en_plot, es_plot, fr_plot...). Eventually, as more languages are introduced to the application, our 'films' table holds over 100 columns, the majority of those related to maintaining data on over 40 languages. Applying vertical sharding, we could create new tables to keep the data in each language (en_film_data, es_film_data, fr_film_data): each of those tables would require only three columns ('title', 'plot', and film_id to maintain the association with the original record on 'films'). With this, we can reduce the size of the 'films' table without reducing the amount of rows.