The Pragmatic Programmers

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?

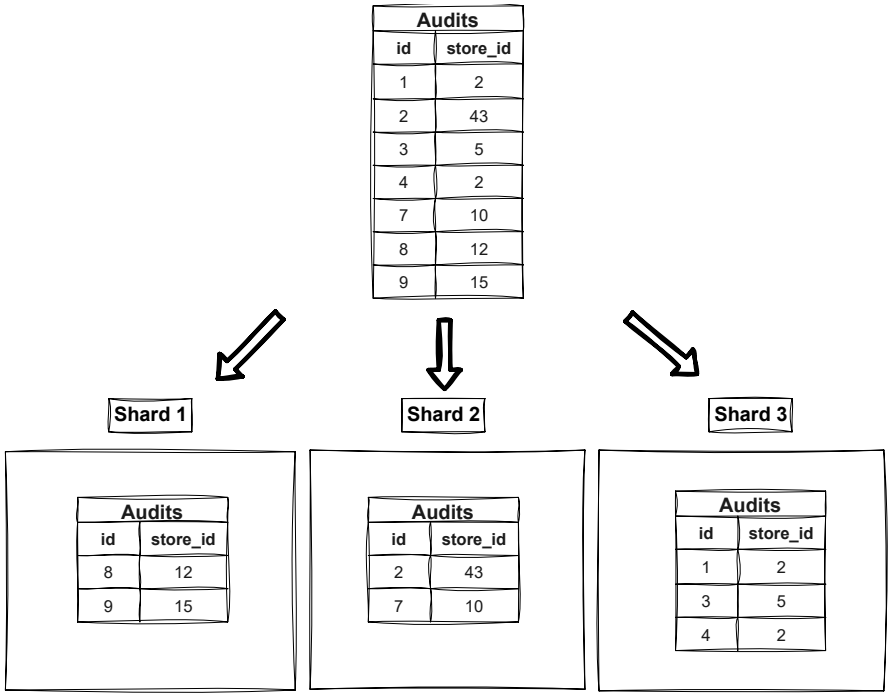
Vertical sharding

Yes, the term "horizontal sharding" implies the existence of "vertical sharding".

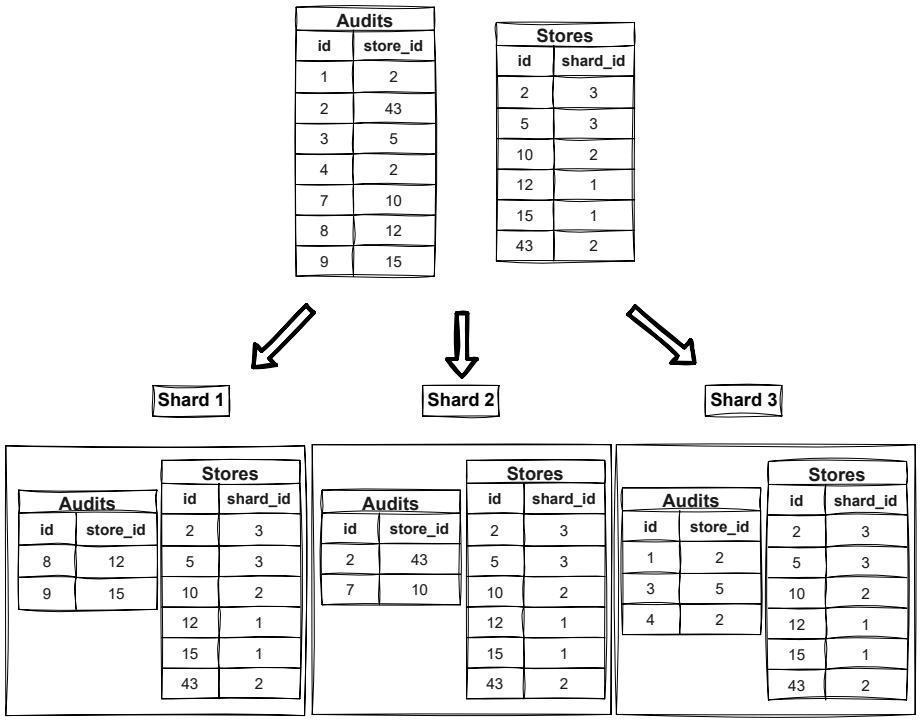
> In computing, a cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests that can be served from the cache, the faster the system performs. \[...\] To be cost-effective and to enable efficient use of data, caches must be relatively small.

The thing is that 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, say, a cloud-based SaaS to manage their human resources, they don’t expect their 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 managing 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 would have 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`. We are going to use this opportunity to add a new feature to our application: we are going to introduce audits. Every time that something of relevance to the store occurs, we are going to create an object and store it. This is how it could look:



What you have just seen described is an ideal sharding scenario. So ideal, that it rarely happens in reality, even in use cases in which data sharding is commonly used. The thing is that, even if all the data introduced by the customer can be completely sharded, in all probability there will be data that is shared. The most obvious 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 a problem.



After this explanation, you may think that sharding is an obvious choice, almost something that you should implement preemptively to 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.