Five sharding data models and which is right

Written by **Craig Kerstiens**

August 28, 2017

When it comes to scaling your database, there are challenges but the good news is that you have options. The easiest option of course is to scale up your hardware. And when you hit the ceiling on scaling up, you have a few more choices: sharding, deleting swaths of data that you think you might not need in the future, or trying to shrink the problem with microservices.

Deleting portions of your data is simple, if you can afford to do it. Regarding sharding there are a number of approaches and which one is right depends on a number of factors. Here we'll review a survey of five sharding approaches and dig into what factors guide you to each approach.

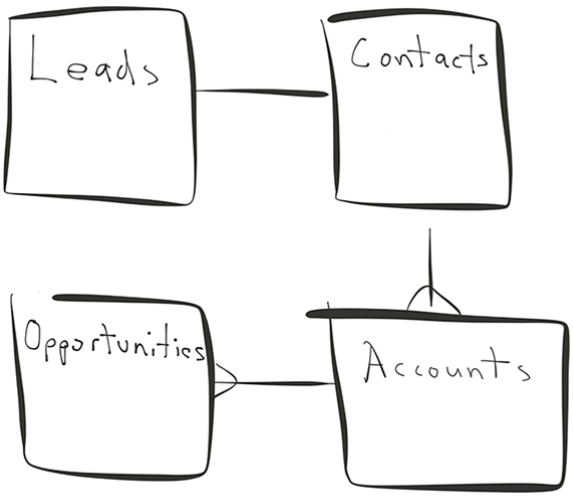
Sharding by customer or tenant

SaaS applications often serve many customers (called 'tenants'), which is why we often refer to these apps as "multi-tenant applications." If you're a SaaS business, it's often true that data from one customer doesn't interact with data from any of your other customers. This is quite different from a social network which has a lot of interdependencies between the data generated by different users.

With a SaaS application that is multi-tenant, the data is usually transactional. It turns out paying customers don't like it too much if you lose some of their data along the way. Due to the nature of many of these transactional, multi-tenant SaaS applications (think: CRM software, marketing operations, web analytics), you need strong guarantees that when

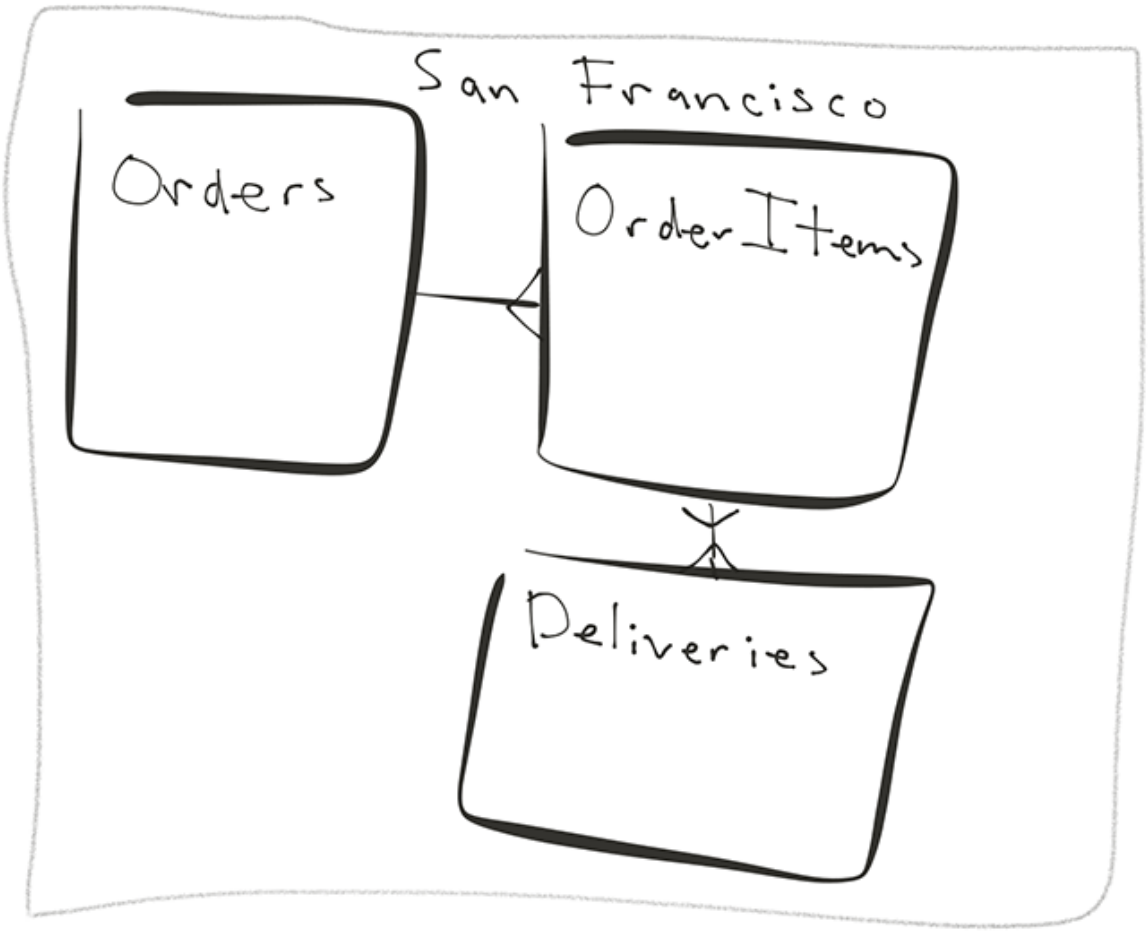
data is saved to the database, the data is going to stay there. And your customers expect you to enforce referential integrity.

Another interesting thing about multi-tenant applications is that their data model evolves over time to provide more and more functionality. Unlike consumer apps which benefit from network effects to grow, a B2B application grows by adding new (ideally sticky) features for customers. And more often than not, new features mean new data models. The end result is 10s to 100s of tables, at times with complex joins. If this sounds like you, sharding by tenant is a safe (and recommended) approach.



Sharding by geography

Another class of applications that has come to the forefront in recent years are location-based apps that key off geography: *thanks iPhone*. Whether it's Lyft, Instacart, or Postmates, these apps have an important tie to location. You're not going to live in Alabama and order grocery delivery from California. And if you were to order a Lyft pick-up from California to Alabama you'll be waiting a good little while for your pickup.



But, just because you have an app with a geographic slant to it, doesn't mean geography is the right shard key. The key to sharding by region is that your data within a specific geography doesn't interact with another geography. Not all apps with geographical data mean a geography sharding approach makes sense. Some apps that require data heavily interact across a defined geographical boundary (such as Foursquare) are less of a fit for sharding by geography.

Similar to the multi-tenant app above, data models for apps that depend on geographic location tend to be a bit more evolved. What I mean is, with apps that depend on location, you have a number of tables that have foreign key dependencies between each other and often join between each other on their queries. If you're able to constrain your queries to a single geography and the data seldom crosses geographical boundaries, then sharding by geography can be a good approach for you.

Sharding by entity id (to randomly distributed data)

Our next data model can lend itself to a number of different database systems, this is because it doesn't have strong interdependence between your data models (read: no joins). With a lack of a need for joins and often transactions in the same sense as a relational database there is a swath of databases that may or may not be able to help. The problem you're solving for is that you have too much data for a single node (or core) to be able to process quickly.

When sharding by entity id,we want to distribute data as evenly as possible, to maximize the parallelism within our system. For a perfectly uniform distribution, you would shard by a random id, essentially round-robin'ing the data. Real life is a bit messier. While you may not have 50 tables with tight coupling between them, it's very common to have 2-3 tables that you do want to join on. The most common place where we see the entity id model makes sense is a web analytics use case.

If your queries have no joins at all, then using a `uuid` to shard your data by can make a lot of sense. If you have a few basic joins that relate to perhaps a session, then sharding the related tables by that same shard key can be ideal. Sharing a shard key across tables allows you to co-locate data for more complicated reporting, while still providing a fairly even distribution for parallelism.

Sharding a graph

Now we come to perhaps the most unique approach. When looking at a graph data model you see different approaches to scaling and different opinions on how easy sharding can be. The graph model is most common in popular B2C apps like Facebook and Instagram, apps that leverage the social graph. Here the relationship between the edges between the data can be just as key in querying as the data itself. In this category graph databases start to stand on their own as a valid solution for social graphs and apps with very high connections between the data. If you want to dig in further, the Facebook paper on their internal datastore TAO is a good read.

With a graph database there are two key items, *objects* which are typed nodes and *associations* which designate the connection to them. For the object, an example might be that Craig checked in at Citus; and the association would be with Daniel, who subscribed to Craig's updates. Objects are generally things that can recur, and associations capture things such as the *who* (who subscribed to updates, who liked, etc.)

With this model, data is often replicated in a few different forms. Then it is the responsibility of the application to map to the form that is most useful to acquire the data.

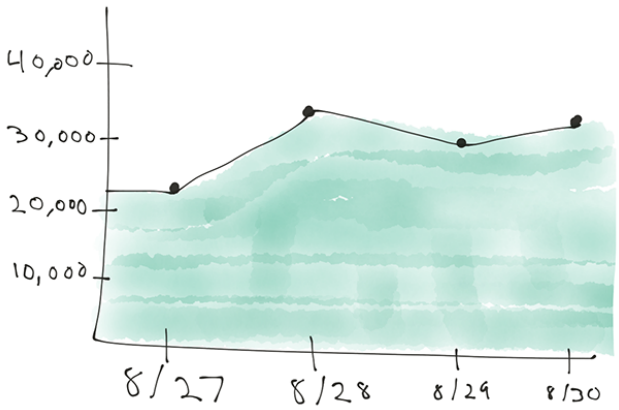
The result is you have multiple copies for your data sharded in different ways, eventual consistency of data typically, and then have some application logic you have to map to your sharding strategy. For apps like Facebook and Reddit there is little choice but to take this approach, but it does come at some price.

Time partitioning

The final approach to sharding is one that certain apps naturally gravitate to. If you're working on data where time is the primary axis, then partitioning by day, week, hour, month is right. Time partitioning is incredibly common when looking at some form of event data. Event data may include clicks/impressions of ads, it could be network event data, or data from a systems monitoring perspective. It turns out that most data has some type of time narrative to it, but does that make partitioning by time the right choice?

You will want to use a time-based approach to partitioning when:

1. You generate your reporting/alerts by doing analysis on the data with time as one axis.
2. You're regularly rolling off data so that you have a limited retention of it.



An example where time partitioning does make sense is when you're an ad network that only reports 30 days of data. Another example: you're monitoring network logs and only looking at the last 7 days. The difficulty comes in when you have a mix of recent data (last 7 days) and historical data, say from a year ago.

The right approach to sharding depends on your application

As with many decisions about architecture and infrastructure, tradeoffs have to made and it's best to match the approach to the needs of your application (and the needs of your customers!) In the case of sharding, matching the type of sharding to the needs of your application is key to being able to scale effectively. When you match your data model and use case to the right type of sharding, many of the hard issues like heavy investment to application re-writes, ensuring data consistency, and revisiting your problem when it's

gotten worse 6 months later can fade away.

Are you trying to figure out how to scale your database—and which type of sharding is right for you? We're happy to help: drop us a note and let's see if Citus is a good fit for you.

SHARE THIS POST

COPY LINK

Written by Craig Kerstiens

Former Head of Cloud at Citus Data. Ran product at Heroku Postgres. Countless conference talks on Postgres & Citus. Loves bbq and football.

multi-tenant popular Postgres sharding time series

POPULAR POSTS

When to use Hyperscale (Citus) to scale out Postgres

By Claire Giordano