# How To Scale Payment Systems With The Sa
# Pattern

What a database paper can teach us about scaling payment systems beyond a s
relational database

ALVARO DURAN
AUG 21, 2024

♡ 11  💬 4  🔁 1

Most times, a single giant database does the trick. At enough scale, it doe
not.

The most common piece of advice I hear to build payment systems is to '
things simple". Often, this takes the form of "don't build distributed syste
It's better to have a single source of truth, because it's easier to understa
system that lives in one node than one that lives in many.

This is good advice. Lots of companies started that way. Nobody, not eve
payments, starts with microservices. Before moving to a Service Oriente
Architecture, Airbnb accepted money via PayPal, and sent checks via mai

Airbnb switched to services when they expanded outside the US. The
pressure that put on their data systems made the switch a business nece

There's a crucial moment for every payment system that scales. It's wher
engineers realize that scaling out a bunch of servers on top of a single
relational database isn't going to cut it anymore.

This, by the way, is something that **many companies never get to experi**

But for some, that's a painful moment. Unless you do something, the syst
will not be able to cope. Payments will not be processed, reports will take

longer to produce, and reconciliation mistakes will be more frequent.

Once it's clear that a single giant database isn't going to be enough, what you do about it?

Most engineers know that NoSQL databases are hard to handle. But they better at this scaling stage than relational databases. That's why Stripe e up using a version of MongoDB to achieve 99.999 percent uptime.

The problem is, good engineers shouldn't go all in. Completely rewriting system to use a different way of storing data isn't a sound idea.

**Good engineers break big migrations in parts. Is there a way to do that databases?.**

In today's article, I'll be exploring a possible way to do that, inspired by tl lessons learned while building Halo 4.

I'm Alvaro Duran, and this is *The Payments Engineer Playbook*. Search ho build payment systems online, and you'll find tons of content on how to software design interviews. But there's not much that teaches you how t build this critical software for real users and real money.

The reason I know this is because I've built and maintained systems that handled payments for almost ten years. And I've been able to see all type interesting conversations about what works and what doesn't for payme systems behind closed doors.

These conversations are what inspired this newsletter.

In *The Payments Engineer Playbook*, we investigate the technology that transfers money. And we do that by cutting off one sliver of it and extrac tactics from it.

When I have a problem that I don’t know the answer to, I gather a bunch scientific papers on the topic, and read them.

So does Caitie McCaffrey.

McCaffrey was part of the engineering team that brought you Halo 4. Bu before they started, it was clear that the way they had handled data befo wasn’t going to be enough.

Millions of users were going to play the new game. Scaling out servers o of a single database wasn't even going to come close to what they neede

But before I tell you what she found, what is so special about serving you users from a single database?

In one word: consistency.

The good thing about relational databases, the reason why most of us sh use them, is that they come with two special guarantees.

On the one hand, you've got **ACID compliance**. ACID is an acronym that down to this: the data will be valid, even in the face of errors and failures the other, relational databases also provide **transactions**, which is a way combine several operations into a single unit.

ACID gives you a way to make concurrent changes to the database safe. transactions let you undo those changes as if they were a single operatio

However, a single database doesn't permit you to separate data to get it to their users across the globe. A single database doesn't allow you to spr out operations to several servers to support more of them. And a single database cannot serve users when some part of it has crashed.

At a lower scale, that doesn't matter. But as your payment system grows, an ongoing source of problems.

Splitting the data across multiple partitions gives you some of that. But you have ACID and transactions if you do that?

Maybe. I'm not a researcher. Perhaps there's something the database community hasn't figured out quite yet.

But not for lack of trying.

In fact, Facebook published a few years ago a paper called Challenges to Adopting Stronger Consistency at Scale. That paper pretty much said tha they were unimpressed by the research done so far. And that Facebook h adopted almost none of it.

In other words, Facebook put itself out of the race to design practical distributed transactions. Google didn't though. But what they've built so called Spanner, is prohibitively expensive.

So far, we've settled for *in the wild* solutions, also called Feral Concurren Control. Engineers like you and me have written a bunch of logic to mov some of the database's responsibility into the application layer.

If you've ever heard of the Unit of Work pattern, that's what I'm talking al

So what Caitie McCaffrey was asking herself was this: **can we do better t feral concurrency?** And while reading scientific papers, she stumbled ac solution on a 1987 paper.

The kick is: it wasn't a distributed systems paper.

And I can't shake the feeling that payment systems are extremely well su for such a design pattern.

Let me explain. The paper is Sagas, and it's a paper from the database community. The authors had investigated a way around the limitations imposed by Long Lived Transactions, or LLTs. Those are the transactions take longer than usual, blocking the completion of the other, faster transactions.

The solution they suggest is to break down LLTs into a collection of smal transactions, which could be interleaved with the rest. Sagas are such collections.

So what's the connection with distributed systems? At the end of the pap there's an intriguing sentence that caught the attention of McCaffrey:

> We believe that a saga [as a distributed system] can be implemented w relatively little effort.
>
> — Sagas

It was as if the authors had left the most valuable part of the paper as an exercise to the reader.

A challenge.

That was the insight that McCaffrey needed. **Sagas were the missing ste between the single relational database and the distributed system** that could make Halo 4 possible.

How do sagas work? **Sagas trade *atomicity* for *availability***. Rather than h a slow transaction locking many tables, you have smaller, discrete action That break down the same process, while exposing intermediate checkp along the way. At any of those points, another transaction can come in a their work. That makes throughput better.

Doesn't that remind you of payments? Isn't money software a sequence steps that the payer experiences as a single flow?

But wait, there's more. What about undoing a saga? How does a "rollback work if you have a bunch of steps that have been already completed?

When a saga fails, the process will call *compensating* actions. These actio correspond to those of the saga. But unlike in a transaction, they undo t effects of their correspondent action **semantically**. What that means is t the undoing is a domain concept. That, unlike transactions, **you don't necessarily go back to the state you were at the beginning**.

Is that bad? It could be. But often, it's OK. If a saga action sends an email, compensating action might be sending a follow up explaining there's bee mistake. Like the way Amazon sends you an email if your payment doesn through.

In fact, the Sagas paper suggests two strategies to deal with failures.

One is called *backwards recovery*. It's what you would expect it to be. If a fails, the process calls all compensating actions for every action in the sa that was completed before the failure. A complete undoing.

Let me give you an example. If, after authorizing a payment, there's some items in an order that are no longer available, the saga that completes th order fails. The compensating actions will remove those items from the i list, and void the payment.

But we can choose another strategy, called *forward recovery*. In this case saga will complete every time. For that, the saga reverts back to a safe checkpoint, and tries again.

Let's take the same example I used before. In a forward recovery, the compensating actions will also remove the unavailable items from the ite list. But, unlike backwards recovery, the saga proceeds with the remainir order, capturing only the amount for the items that made it all the way through.

I know I'm talking about a way to make payments distributed. But even fc payment systems that rely on a single relational database, this pattern is useful.

That's because **sagas are a failure management pattern**.

They help engineers think about failure modes more proactively.

I've hinted a bit so far about what would it take for payment systems to implement sagas, so here's an overview of what I would expect to see in a actual implementation:

- **A Payment Saga would consist of at least three actions**: Let's call th authentication, authorization and confirmation. The reason I say *at l* *three* is because these actions will most likely consist of more subste such as fraud checks, 3DS, waiting for a pending action to be notifie complete by the provider, etc.
- **Each of these actions will have a compensating action: Let's call the rejection, void, and refund.** Something worth noticing is that the ref compensating action will undo both the confirmation and the authorization, and therefore the void step must take into account th payment could have already been refunded by the time it gets proce
- **A recovery strategy:** Forward, Backwards, or a mix of both dependin the use case.

If you need more guidance on how to build sagas for payment systems, l know in the comments. For now, you can check McCaffrey's talk on Distributed Sagas.

So let me wrap this up. Modern Treasury CEO Dimitri Dadiomov once w that reversibility and idempotency are the two specific engineering conc that anyone building payment systems must keep in mind. The good thin about relational databases is that they provide you with both out of the b

---

> *Scaling a payment system beyond a single database is accepting the responsibility for keeping those guarantees intact.*

---

That's why a database pattern is so suitable for building distributed syste

It's not the technology that matters, but what you use it for.

And that is why distributed transactions are too expensive and impractic Scaling operations come with unavoidable trade-offs. **What do I *really* n and what can I *do away with* to get it?**

And that's my biggest takeaway from this article.

Engineers often rely on what others have done to do their job. That's a g thing: Newton called it [standing on the shoulders of giants](). However, the cost to not fully grasping what we really need from the tools we use, and are simply "good-to-haves". And that cost is that, if you're forced to aban such tools, then you feel the compulsion to recreate them as faithfully as can.

You are better off if you choose to examine what your new scaled life demands from you, tool-wise.

And exploit a new set of trade-offs to achieve it.

That's it for *The Payments Engineer Playbook*. I'll see you next week.

Thanks for reading The Payments Engineer Playbook! Subscribe for free to receive new posts and support my work.

Type your email... | Subscribe

**PS**: Before you go, you've made it this far, just give me 30 seconds.

I've got good news.

The feedback on the last 3 articles has been overwhelmingly positive. Pe had commented on my LinkedIn posts, shared the articles at work, and emailed me with words of appreciation and support.

Even Charity Majors said that my article on Uber brought her joy.

It felt awesome. I've tried to answer every comment, DM and email, and I
sorry if I didn't reach out to you in particular. I promise I will.

So the good news is that I'm going to keep doing the amount of research
I've put up with to give you more of these articles.

But I'm only going to keep doing it if you let me know that they are usefu

So please, if you're on LinkedIn, go to this article's post and leave a like, o
review, and let me know what you thought of it.

Give me your feedback.

And also, by the way, if you hate these things, you can let me know too. I
to hear from you.

Should I keep making these? Because they take way more work than you
imagine, and I only want to do them if you absolutely love them.

By giving it some love on LinkedIn, The Algorithm knows that the conten
valuable. And it gets shown to more people.

And if you got this article from a colleague, do me a favor and subscribe.
week I feel I'm getting better at this. That means that my best articles on
to build payment systems are probably yet to be written.

You can only find out if you subscribe to *The Payments Engineer Playbook*
see you around.

---

11 Likes · 1 Restack

← Previous Ne

## Discussion about this post

Comments Restacks

Write a comment...

Manju Easwaran Aug 21, 2024

Liked by Alvaro Duran

Exactly what I was looking for! Great insight into the large scale payment processing issue

LIKE (1) REPLY

**1 reply by Alvaro Duran**

Sunday Adeyemi Aug 21, 2024

Liked by Alvaro Duran

This is interesting

LIKE (1) REPLY

**1 reply**

**2 more comments...**