EPISODE 911

[INTRODUCTION]

[00:00:00] JM: Modern databases consist of multiple servers that host the data in a distributed fashion. Using multiple servers allows a database to be resilient to the failure of any one database node, because copies of the data are shared to other servers. A multi-node setup also lets the database grow beyond the size of data that could be hosted on a single node.

Although a distributed database gains in scalability and resiliency, a database that runs across multiple nodes has a variety of problems that are not faced by a database running on a single node. Every operation within a distributed database becomes more complex than the single node database.

For example, if you make a query to your distributed database, you might not be able to rely on the answer that you get from any single database node, because the other database nodes might have been involved in transactions that have not yet propagated to all of the nodes.

Alt Cabral is a lead product manager at MongoDB and coauthor of a paper on Causal Consistency in MongoDB. Aly joins the show to discuss the engineering of distributed databases and her experience architecting MongoDB.

Full disclosure; MongoDB is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:01:30] JM: This podcast is brought to you by PagerDuty. You've probably heard of PagerDuty. Teams trust PagerDuty to help them deliver a high-quality digital experiences to their customers. With PagerDuty, teams spend less time reacting to incidents and more time building software. Over 12,000 businesses rely on PagerDuty to identify issues and opportunities in real-time and bring together the right people to fix problems faster and prevent those problems from happening again.

PagerDuty helps your company's digital operations run more smoothly. PagerDuty helps you intelligently pinpoint issues like outages as well as capitalize on opportunities, empowering teams to take the right real-time action. To see how companies like GE, Vodafone, Box and American Eagle rely on pager duty to continuously improve their digital operations, visit pagerduty.com.

I'm really happy to have PagerDuty as a sponsor. I first heard about them on a podcast, probably more than five years ago. So it's quite satisfying to have them on Software Engineering Daily as a sponsor. I've been hearing about their product for many years,and I hope you check it out at pagerduty.com.

[INTERVIEW]

[00:02:57] JM: Aly Cabral, welcome to Software Engineering Daily.

[00:02:59] AC: Hey! It’s great to be here.

[00:03:01] JM: Yes, it's great to have you. You are a distributed systems product manager, essentially. Although, we'll get into that in a little more detail what that actually means. Distributed systems theory is hard. I have not actually implemented to many distributed systems, but I've talked to many people about them. My sense is that distributed systems in practice are even harder. Why is it hard to build distributed systems?

[00:03:28] AC: Most of the time it’s because there're so many variables that affect guaranties in distributed systems. People don’t understand that most things are distributed systems and they’re making tradeoffs every day, whether they think about them hard or not. Now, sometimes there's actually a lot of interesting idea, because in academia a lot of people would think that no application could be successful without linearizeability, right? Which is like one of the strongest consistency guarantees you can give an application.

Now, and fact of matter, there is just so many applications that are successful with lesser consistency guarantees. One of the reasons why people hypothesize that that's the case, is because people don't notice when those guarantees are broken, or they’re so rare. Failures in the system are so rare that you're willing to kind of eat the pain when a failure goes wrong. So it's really an interesting area for applications that are critical in the sense that they can never have these guarantees broken. There is no room for an inconsistent read, for example. What kind of the guarantees you can give them without them having to sacrifice performance, right?

So MongoDB, we've built this kind of tunable consistency model that allows people to make these tradeoffs an at operation level, right? So that not every application has the same weight or not every operation in your application has the same weight. So you can decide whether to pay the latency cost of specifying strong consistency at the time you need it. It can be really challenging to think through all of these variables.

[00:05:00] JM: Right. When you think that you have thought through all those different variables, is there a way to actually test a distributed system to ensure that it works the way that you think it works?

[00:05:14] AC: So there are couple of frameworks out error. One of them I think probably the most popular is that Jepsen test suite. Now, the Jepsen test suite actually is pretty tailored to the products or databases that they've tested, and they create a suite based on the guarantees that the database has claimed to have given them, the Jepsen test suite. Per product, they decide to kind of evaluate or audit.

Now, there is no really generic framework, because a lot of people specify the guarantees they give in very different flavors. There's not really standardized understanding of these guarantees across the board that are largely testable in a like drop and play kind of scenario. But Jepsen is a really pretty well- respected within the industry. It’s a place that people look to.

[00:06:02] JM: When you want to implement a construct from a distributed systems paper, when you want to implement something from the distributed systems literature. How straightforward is it? Does the literature typically cover the edge cases in the practical implementation details of a distributed system or do you have to figure out a lot of it from your own tinkerings?

[00:06:28] AC: So in most case, literature is fantastic for building the foundation of what guarantees you want to provide end users. Now, often you have to go beyond that to determine – Like save users from themselves. So determine a way to give them that guarantee or behavior without also allowing them to shoot themselves in the foot.

Now, academia doesn't have to worry about behaviors for users that aren't distributed systems experts, right? Everyone in the room is the distributed systems expert and has PhDs in this and thinks about these problems really hard. So they're not thinking about how to make these guarantees accessible to everyone, right? That's not the types of problems they're trying to solve.

So, often, you'll have to take a theory from academia and then make sure that we make it accessible to users so that they aren't spending their fulltime job thinking about distributed systems, because then you're not doing a very good job of building a product that makes them more productive.

[00:07:26] JM: There are different kinds of distributed database. So distributed databases can make variable tradeoffs in terms of scalability or consistency. Describe some of the kinds of tradeoffs that different distributed databases can make.

[00:07:43] AC: Yeah. I'll give one that's an example. There are multi-master systems and their single leader systems. MongoDB happens to land in the single leader system category. Now, in a multi-master system, it's hard to have an order of events, right? So you have multiple nodes that are accepting writes at the same time. They are accepting these writes. It's hard to order these writes at the end of the day.

Now, we always talk in talking distributed systems like there's no such thing as wall clock time, because every machine has its own perception of what the time is. So you kind of have to rely on this concept of logical time. So every modification that comes into the system increments a counter, and that's how you increment time. That's how you progress forward.

Now, in a multi9master system, you have everyone kind of having their own notion of the order of operations in a system, right? Eventually they'll go in apply these sort of operations across the board. Now, that kind of system makes it challenging to provide guarantees like causal consistency guarantees, where you want to read or guarantee that you've read your own writes.

In this case, what you might do is wait for a write to be committed to a majority of nodes. Then on the read, also wait for a majority of nodes to have replicated that read and make sure that you have a coordinator that like coalesces all of the responses from majority of nodes. You have to actually go and outreach to them and then return responses once you've heard from the majority. So you have to pay both on the write and the read latency, because you can't rely on an order of events. Like there's no central orderings. You have to at every time reach out to each of those nodes.

In a single leader system where you do have a notion of order, because you have a single leader that determines the order of events, you can, as an operation, like as a client-side operation, keep track of the time you're write was applied. So when you get a response from the leader, it has a time associated with it. That's that logical time. Then return back to a secondary or when you're doing a read from a follower, or a secondary in our language. Then you can pass along that time and say, "I do not want to hear from you until that time has been applied locally."

And in that scenario, you actually don't have to reach out to any other node. You can do that operation locally. So those are some of the tradeoffs. If you have a system that is ordered or unordered, you can provide some more performant guarantees at the stronger side of things.

[00:10:13] JM: We'll get into some implementation details of how you make some of these guarantees around consistency via clocks. But just talking about a high-level. So you work in MongoDB. You're a product manager and you – It sounds like it's a very technical product manager kind of role where you have to make certain technical decisions around database consistency. Can you just tell me what is your interaction with the different teams at the company and how does your role take place?

[00:10:54] AC: Yeah. So the role is very technical, but the product management role is really about representing users and customers when we're making these decisions, right? We could do what we do and make decisions and isolation from the engineering side. But engineers aren't spending a lot of time talking to customers or understanding how applications work. We have a lot of systems engineers that are amazing at building these distributed systems but don't really empathize with building applications that leverage distributed systems, right?

So I'm meant to represent that perspective so that we build well-rounded products and not products that people don't know how to use or aren't useful in the practical sense. So it's about marrying what we know we're strong at in theory and building something practical that people can use. That's really what the role is. Now, how does that work in practice?

Well, I work very closely with understanding our users and our customers. So I will go understand what applications need and what they want to go onsite with our users and spend a lot of time building up that gut instinct about what users need, because that's really the benefit I'm providing in this decision-making process from an architecture perspective. Like when we're weighing different architectures against each other or are different approaches, I'm meant to provide that unique value of how is this useful in the wild.

Now, I work very, very closely with the engineering team when we're deciding what to move forward with from a design perspective so that we land on something that is both feasible to implement and useful across the board. How that works for us is that I'll be involved in like the scoping and the design and the specking of any feature that we deliver that's user-facing and effects guarantees their behavior for a user.

[00:12:49] JM: How do you have conversations with users about distributed systems guarantees? What kinds of users are these and how are they sophisticated enough to give feedback into like low-level database consistency questions?

[00:13:04] AC: Right, and the truth is you often don't have distributed systems conversations, right? We have conversations about applications and the behavior they want for their application. What they would consider a bug, I would walk them through edge cases, like what would the application do in this scenario. If their answer is I haven't thought of it. Well, then we better not provide them that scenario, because they're going to like have a bad experience if that scenario comes up in the wild for them. So we don't have often distributed systems conversations with our users. We have conversations about applications that I can then apply to distributed systems decision-making.

[00:13:39] JM: Interesting. Well, let's talk about MongoDB specifically. So MongoDB is widely used is a transactional database. It's often forming the backbone of an application. This is a particular use case that differs from something like a time series database or a distributed queue. How does the use case of a transactional database that's forming the core application backbone? How does it affect the architecture, the distributed systems consistency properties that you want out of the database?

[00:14:18] AC: Yeah. So there's a couple reasons why MongoDB as a distributed system really took off the way it did when we started kind of building these modern applications. One of the reasons is that users have never been more impatient than they are today, right? That's just the truth of the matter, right?

We used to be okay maybe 20 years ago, 30 years ago with an hour of downtime or two hours of downtime. Now, people don't want any downtime. They get upset if they're waiting second for a second to load, where we're in the world where milliseconds are the expectation. Response time matters. That's just what our users expect. We've involved to really be at a point where you have to kind of meet these very, very strict SLA's both for downtime and responsiveness of the application.

Now, if the data isn't accessible, often, like if you can't read or write to the data. That means your application is down often. That's really bad from a transactional application perspective. That means you don't hit those user needs. You're not heading your SLA's.

Now, this is not the part of the system often that is your business differentiator from application perspective, right This is not the thing that is going to give you an edge over a competitor. But it is the thing that is critical to maintaining a good user experience with your end users, but you don't want to spend a lot of time thinking about it, because it's not the thing that's giving you a competitive edge versus the competitors.

Now, how MongoDB helped with this is a couple of things. One, with replication. So now if you have –I'm going to call it a primary. Before, I referred to this as a single leader system. So if you have a primary that accepts or reads and writes, you can now have any number of secondaries that replicate that same set of data so that if the primary goes down, you still have the redundancy. You can have an automatic collection and you don't have that downtime of an hour, right? There is no single note system anymore. You have redundancy. You have automatic failover, and that really gives you the ability to address the expectations of your users.

Now, the second thing that MongoDB offers as sharding, and that is the ability to partition data in a cost-effective way. So when relational systems were built, we lived in a world of mainframes, like these big, big machines. In that world, if the mainframe went down, your database went down. But the mainframe never went down, right? It was a very reliable piece of hardware that costs millions of dollars. That kind of model was okay, both because the mainframes were reliable and our users were a little bit more lenient about the downtime that they were willing to accept.

Now, in this model where we have a lot of commoditized hardware, smaller boxes spread across multiple, maybe even cloud provider, multiple instances, there is more cost effective way to scale. Now, in the mainframe days, let's say I needed a mainframe that supported 2X the workload. Well, that could be 10X more expensive, right? That's not a linear cost growth, right? It becomes exponentially more expensive the bigger the machine you buy.

The commoditized approach in the cloud world, the more machines you add are linearly more expensive, right? If I have a machine with one core and add another machine with one core, that's just twice as expensive, but I'm getting twice the amount of resources. So it's a cost-effective way to scale at the large-scale of data that we have today.

Now, sharding is our way to partition the data across these commoditized hardware in the cloud world that doesn't require the application to change any queries, change any logic. If you add a shard or add a new machine, we would automatically start adding data and leveraging that machine to kind of take advantage of the hardware you throw at it. So those are the two core aspects of distributed systems in MongoDB that gave MongoDB quite a bit of popularity in the early days. Kind of it came at the right time.

[SPONSOR MESSAGE]

[00:18:36] JM: GitLab Commit is GitLab's community event. GitLab is changing how people think about tools and engineering best practices, and GitLab Commit is a place for people to learn about the newest practices in dev ops and how tools and processes come together to improve the software development lifecycle.

GetLab Commit is the official conference for GetLab, and it's coming to London October 9th at the Brewery. If you can make it to London on October 9, mark your calendar for a GetLab Commit. Go to softwareengineeringdaily.com/commit and sign up with code COMMITSED to save 30% on conference passes.

If you're working in dev ops and you can make it to London, it's a great opportunity to take a day away from the office. Your company will probably pay for it. I'm going to go to a GetLab Commit at some point. It won't be this one in London unfortunately. But I will definitely go, because I'm excited about the GitLab ecosystem. It's quite an interesting ecosystem. Seeing it develop has been cool over the course Software Engineering Daily's. At GetLab Commit, there are speakers from VMWare, Porsche and GitLab itself. So if you're in London on October 9th, you can check it out. I hope you do check it out. Thanks to GitLab for being a sponsor.

[INTERVIEW CONTINUED]

[00:20:11] JM: You've given some database history, and I've had some conversations with some database people where when they talk about the database problems of the early 2000's, for example, so like I've had conversations with people who were scaling relational databases at YouTube and PayPal, and they moved to this model of partitions and replicas. So you break up your database into partitions and then you replicate each of those partitions so that if one of the partitions dies, you have replicas the you can serve.

But one of the early problems that I think happened as people were moving to this commodity hardware was that although you would have much more cost-effectiveness, because you would have to address these different partitions, or maybe even in some cases the replicas of those partitions, you would have to have your client, your application, for example.

Literally, your client would have to know where to go to find the right partition. So rather than saying, "Hey, database. Give me you select star from whatever." You would have to say, "Go to this partition and then select the data from this partition," which created more overhead for the client. Do you have any knowledge of whether Mongo had that requirement where you had to address the replicas or addressed the partitions in the early days? I imagine that if so, it's been ironed out at this point. But do you know anything about the client addressing?

[00:21:58] AC: Yeah. In MongoDB sharding, which is our word for partitioning. So I'm going to use those interchangeably. So when I say shard, I mean partition. So in MongoDB's version of partitioning or sharding, we have the concept of config servers. Now, config servers are replicas. They are also redundant, that have the map of where the data lives, right? They have the chunk map. Chunk maps are logical ranges of the dataset.

Now, in MongoDB's sharding, we've always had this concept of a router, the MongoS. This router caches the chunk map from the config servers and then routes queries accordingly. So you can add a new partition, and it's completely opaque to the application. The application makes no changes. The router is the thing that's updated as we change the chunk locations or do a chunk migration. We can dynamically change the location of these chunks as soon as you add a new shard and it will be completely opaque to an application.

[00:23:03] JM: Right. Now, as we're talking about the MongoDB cluster itself managing the locations of different chunks of data, we can start to realize that if a right is issued to this MongoDB cluster, the cluster is going to need to figure out how to manage the consistency properties of the data that's getting written to the database. Here we can start to talk about the implementation details of MongoDB's consistency. So a distributed database needs to be consistent. One way to maintain consistency is through a clock system. Can you explain the role of a clock system in maintaining consistency?

[00:23:56] AC: Yes, absolutely. So clocks are funny in distributed systems, where any node at any given time may have a different notion of physical time. So whether I have a machine in EMEA, versus North America, versus in Asia, they might all be configured to have roughly the same time. But in practicality, maybe they're off five seconds. Maybe they're off by hundred milliseconds.

Now, there are tools in which you can minimize clock skew, right? So make sure that all of the clocks are synchronized across your distributed system. But MongoDB actually doesn't guarantee that you're using a tool like, right? So we can't reliably trust a physical time.

So in distributed systems, a common way to overcome an untrusted physical time is by using a logical time. Now, this implements the same kind of clock. One of the most cited papers in distributed systems is a Lamport clock, which is just a very, very simple incremental clock that increments every time a modification comes into the system. So if you have no modifications happening in your system, it might as well be the same time. So even if i don't have any modifications that happen in the last physical day, that would anything that happens during the day, any read that's executed during that day were no modifications are happening would be executed at the same logical time.

Now, as much modifications come into the system and it's possible to get different results, then that's how we're incrementing this logical time. So in distributed systems, you really can't trust physical time. We rely on some of the concepts that we derived from academia around logical time. There are some benefits to still keeping around the physical time. So in MongoDB, we've actually implemented a hybrid logical clock based on the Lamport clock.

[00:25:50] JM: So this may sound strange to people who have not read much of this literature. The idea that you could have a clock that is not like a normal clock. It's not like you look at a clock on the wall and you can trust that that clock is telling you the time and the time that everywhere everybody in the same city or the same country is agreeing on. This is more of a relative sense of time. Can you describe the idea of relative time, which is achieved through vector clocks or Lamport clocks?

[00:26:32] AC: Yes. So as we've all experienced probably, your phone often can be maybe off of the time that's presented on your computer, right? Maybe it's off by a minute. Maybe it's off by 30 seconds. But there is some notion that the clocks that you're seeing across the pieces of hardware that you have are slightly off.

Now, in our practical world, the amount it’s off barely matters, right? You're going to make the meeting on time regardless of whether your computer is 30 seconds ahead of your phone. Now, in distributed systems world, where there are multiple writes happening and maybe subsequent reads that want to read those writes, you actually have to have a stronger notion of relative time.

Now, in a sharded, like a partitioned system, this gets even more complicated. Now, the reason why is because each partition is accepting its own set of rights, right? Completely separate to a different dataset. It’s having its own separate sense of time, right? It's just going along its own timeline and it doesn't really need to communicate proactively to other shards, unless you need it to.

Now, if a write happens on your first shard or a partition and then a write happens on your other shard and partition, how do you correlate what write happened before the other, right? We can trust that physical time. So how do you reliably know the order of events that happened in the system? And the order of events really matters when we talk about consistency guarantees.

Notably, one good example of like why it matters, is let's say I want to stream all of the events that happened in my system and maybe like apply it to a cache, right? So I want all of the recent rights in my system that I can then update my cash with. Now, I have a process that tales MongoDB. Streams all of the events that are happening. You would use in MongoDB this feature change streams.

But let’s say that process goes down for a period of time, right? It's down and then it needs to come back up online and resume, right? I have this cache to refresh. It’s not going to go away. How do I resume that stream? Now, in this sense, I need to have a concept of relative time, because otherwise I might miss events if i don't know where to pick off from.

So if I don't have this total order of events, I wouldn't be able to resume the latest changes in the system and start applying them from there. So that's a good reason why relative time is so important in a distributed system and why this order of events helps us understand relative time across partitions.

[00:29:20] JM: Another way we could define this term is causal consistency. Could you describe what that term is, causal consistency?

[00:29:29] AC: Yes. So causal consistency is often defined as the behavior, like the user behavior of reading forward in time. So let's say that I read from a replica, right? Let's say I'm replicating the same set of data across five nodes, and I read from one of them. Let's say send another read request to another one of them.

Now, I could actually target a scaler node that hasn't replicated up to the note that I just read from, right? So it has a scaler version of the data. Now, if I were the target on my second read request a scaler version of the data, that would be considered reading backward in time, right? So like using a simple example here, let's say I'm a user. I update my username. So I write to the primary. I update my username, and that data will be replicated to all of the nodes. But that replication takes time, right? There's physics involved here. It's going to take some time.

Let's say I immediately do a subsequent read of my username. Now depending on which node I target, I might see my new username or my old username. Let's say I see my old username. As a user, what I might do is just trying to change my username again maybe. But that's actually an unnecessary operation, because eventually all of the nodes will replicate that new username anyway. I just happen to target a scale node. But I got some weird user behavior because of it. So that is one of the things that causal consistency guarantees you will not do. You will not see back in time. So I, as a user, if I update my username, I will see it on all subsequent reads. I'm not going to read backward in time.

Another way people look at it and another aspect of the guarantee is read your own write, so that if I performer a write, I guarantee that it will see that write somewhere and then monotonically increasing writes, and then writes follow reads. Reads follow writes.

[00:31:33] JM: You have written that causal consistency is not widely used in the database industry. Can you explain why it's not used more widely and what the alternatives to causal consistency are?

[00:31:47] AC: So on a single node, in that legacy world, in that single node worlds that people had in their mainframes, they actually got causal consistency by default, right? If I wrote to the data, I was going to read that data. There was one source of truth. But also one failure point, right? We've changed from that model. But in that model where you have a single node, you get these guarantees, right?

So it's a relatively new concept as the prevalence of distributed systems came about that we've had to kind of think of ways to tackle this on the distributed systems front, and it's really – MongoDB having been a products that treats distributed systems as a first-class citizen. We're one of the first people to bring it as a first-class citizen, bring this guaranty as a first-class citizen to the database without any application changes needed.

[00:32:38] JM: And if I have a distributed system, distributed database, what are my alternatives for maintaining consistency other than this causal consistency, or do i just have to sacrifice? Is this the best practice? Otherwise, I'm sacrificing consistency?

[00:32:54] AC: There are plenty of use cases that can deal with that intermediate behavior that it takes to replicate writes across nodes in a replica set. Most of the time, we're talking milliseconds of inconsistency and not seconds depending on your lag here between nodes. Now, for critical systems, there are people that want this guarantee all the time. I would say those are rare, or at least rarer applications that need those guarantees literally always.

Most of the time, these inconsistencies will manifest in points of failure, and failures are both rare and people can kind of wrap their head around it. If you get inconsistent behavior, when half of your nodes are down, people might be more willing to accept that, "Okay. I don't need to build a system that is for something that happens once every 10 years or once every five years," let's say. So often these inconsistencies will manifest first not for long and then only in failure scenarios.

Now, what are some workarounds? In MongoDB, if you read just from the leader or read and write just from the leader, most of the time you'll be getting this guaranty. Again, except for in cases of network partitions or when you have a lot of elections happening in your system. Some of those guarantees can be broken. But often you will get this behavior if you're just reading and writing from the primary in MongoDB, and you can do something similarly with other systems.

[00:34:28] JM: Well, I think about something like a logging server, or like a time series database. Those are databases where you probably don't need the level of consistency that you need out of a transactional database. Because of I'm just writing log messages, it's not a big deal if I read some log messages out of order. Those are not pieces of data that you're being served to users, where I need to like tell those user, “It's not like you're looking at your banking statements and you need to know these things are appearing in the order in which they happened.”

Log messages can be presented more inconsistently, or time series records. If you're taking the temperature every 500 ms, it's not a big deal if you get some data points out of order, because these things are probably just going to be rendered on a graph somewhere. So that's kind of an illustration of the differences between a database like Mongo and something like time series or log database where you need less stringent consistency.

So you worked on this paper about causal consistency in MongoDB. What was the motivation for writing a paper about implementing causal consistency?

[00:35:46] AC: Really to give back to the community that helped us build the foundation for the feature, that was largely the motivation. We actually presented that paper at SIGMOD earlier this year, which is an very big database conferences that we participate in, and we gained so much knowledge and so much insight from reading papers. We felt we were obligated to get back our learnings from implementing it in a large-scale distributed system that users use.

[00:36:16] JM: And when you're implementing this causal consistency model – So in the paper you talk about clocks. You talk about the construct of o'clock. Do you have like on each of the MongoDB nodes, is there like a process that is called the clock process? What is the object that is representing this relational clock?

[00:36:43] AC: So we keep track of this like relative logical clock within the oplog, which is in operation lot, which contains – It's a capped collection, which is a set size and contains the last modifications that happened in the system. Each one of those modifications has a time written down that's associated with it. We also have just like in-memory cache the last applied time on each node.

Now, we talked about how replicas will replicate the modifications and update to their local notion of their time as their updating their modifications. Now, we use the operational log, the oplog, to replicate that same exact data item in an item-potent way across the systems. Now, we just keep track of that time there, and it's our order of events.

[00:37:33] JM: The causal consistency model that you worked on implementing, was that a replacement first some existing consistency model in MongoDB? I mean, you must've had – There must've been some way of ensuring consistency in the past, and this was a newer consistency model that you implemented. Tell me what you were replacing.

[00:37:55] AC: Yeah. So causal consistency is actually very complementary to the rest of our consistency model. So we have concepts of read concern. It's really equivalent to rad isolation in other systems where you can pair like read concern local or read isolation local, which is basically reads would be able to return the local view, a local copy of the data. Then read concern majority, which is reads can only return the majority committed data to the durable data in the set. So we have these concepts of rate isolation where you can tradeoff like how current the data is versus how durable the data is and the read response. You can actually pair that with causal consistency. You can actually use those concepts together. So causal consistency was often complementary to the rest of our consistency offering in MongoDB.

[SPONSOR MESSAGE]

[00:38:56] JM: Monday.com is a team management platform that brings all of your work, external tools and communications into one place making cross-team collaboration easy. You can try Monday.com and get a 14-day trial by going to Monday.com/sedaily. If you decide to become a customer, you will get 10% off by coupon code SEDAILY.

What I love most about Monday.com is how fast it is. Many project management tools are hard to use because they take so long to respond, and when you're engaging with project management and communication software, you need it to be fast. You need it to be responsive and you need the UI to be intuitive.

Monday.com has a modern interface that's beautiful to look at. There are lots of ways to use Monday, but it doesn't feel overly opinionated. It's flexible. It can adapt to whatever application you need, dashboards, communication, Kanban boards, issue tracking.

If you're ready to change the way that you work online, give Monday.com a try by going to Monday.com/sedaily an get a free 14-day trial, and you will also get 10% off if you use the discount code SEDAILY.

Monday.com received a Webby award for productivity app of the year, and that's because many teams have used Monday.com to become productive. Companies like WeWork, and Philips and Wix.com. Try out Monday.com today by going to Monday.com/sedaily.

Thank you to Monday.com for being a sponsor of Software Engineering Daily

[INTERVIEW CONTINUED]

[00:40:46] JM: What are some general principles that you learned about implementing and applying distributed systems when you were working on causal consistency?

[00:40:55] AC: Well, as you mentioned in the beginning, distributed systems are incredibly hard, and most application developers don't want to spend all of their time thinking about distributed systems. So it's super important that behaviors are intuitive.

But the strongest consistency guarantee often are the weakest performance, and the performance characteristics are incredibly important to a lot of developers as well. So it's been really tricky to manage the tradeoff of performance and the behavior that makes developers life's easy, and that really has been something that I've learned a lot about and I'm continuing to kind of evolve my understanding on as we go through and build new things.

[00:41:45] JM: I can imagine, because if you're changing the engine of the airplane while it's in flight, which is essentially what you're doing by changing the consistency model or improving the consistency model at a low-level. If you impact performance at a low level, that could really have some problematic downstream impacts, because MongoDB has a large user base. So how do you ensure that that doesn't happen?

[00:42:17] AC: Yes. So it can be very tricky, because the obvious answer, like the easy answer, is add a new knob and people can opt into it. The problem is we just talked about distributed systems are hard. No one wants to think about these problems really hard. If you give them a knob, now will come, right? No one's going to use the knob.

So we have to kind of think about how to make these concepts accessible in a way that makes sense from an API perspective. Now, with causal consistency, what we did is we needed the default for any sessions that are opened in the system. Now, we implemented the sessions API at the exact same time we implemented causal consistency. So there wasn't pre-existing users that were expecting certain behavior of sessions and are now broken, because they've gotten 10X lower, right? That's not how this worked in the system.

So we introduced the sessions API, which allows you to associate some number of operations with the session and causal consistency on by default, meaning that all of those operations executed within the session are now causally consistent as a guarantee if you specify the right durability and read isolation as well. So we kind of try and make it intuitive in the API so that people don't have to think super hard about it while maintaining the behavior that people love in their current application.

[00:43:38] JM: Okay. So is that to say that this is essentially an opt-in consistency feature?

[00:43:44] AC: For sessions. Yup, exactly. So sessions, API, it's on by default, but you have to write a session in your application.

[00:43:50] JM: Okay. Interesting. What's an example of an application that would want to use that session API?

[00:43:57] AC: So there are a lot interesting use cases for sessions. The sessions basically allow you to group any number of operations that you're doing. So you can say these three writes, these three reads are all within a section. Basically they're like logically happening in sequence. That's what the API really outlines for a user. So people who want causal consistency guarantees, but don't know the term causal consistency would use a session. Also, sessions are what we built our multi-document transactions on top of. So people who are doing multi-document transactions would also just leverage the sessions API and thus causal consistency too.

[00:44:39] JM: Do you have maybe like a higher-level use case that comes to mind, like an application level use case just to make it tangible for the listeners?

[00:44:49] AC: Yeah, absolutely. So for an example of posting, let's say I want to post to a review site. So let's say that I perform a write, which is a new review in the system. So from the application's perspective, a session is associated with an application instance and really with a user and end user for that application.

So in that session, I my first post the review, which is a write, and then read the new score of the new like aggregate score of the thing I'm reviewing. So let’s I gave it one star and there's two reviews and the other one was a three star. Now, the average rating has gone to two. So I might do an immediate read right after to update the average rating. So any kind sequence of events that need to happen from a user's perspective could happen in quick succession within a session.

[00:45:45] JM: How does security factor into an implementation of causal consistency?

[00:45:50] AC: Yes. So this was really the thing that we had to expand upon from what we were learning from academia. One of the things that we found pretty early on is that if you have a malicious client that is incrementing time too far in the future. At some point, it's going to overflow.

Let's say a 64-bit integer that is incrementing. Now, if someone just does a malicious client or like a malicious node, just increments that time by saying, “Oh! I applied a write at this time,” which is like the max time minus one and overflows the system, meaning that that counter can no longer be incremented. Well, that means that, no, writes can happen. No modifications can happen in the system, because updating logical time is on the critical path.

So I talked about a malicious client now, but this can also be a misconfiguration issue, right? Let's say that I wasn't managing the system properly or I made a dumb mistake. Well, if I accidentally incremented that time, there is no going back. There's no giving me more time. Time only progresses in the system.

So we really wanted from a security perspective to prevent people from making a mistake or maliciously updating that time such that basically the counter would overflow and then no writes would be able to happen in the system, right? We didn't want that to happen.

The way we did that without sacrificing performance, was signing key ranges of time and only the primary of the partition could increment time and it would sign it with its key and then everyone else is just a passive gossiper of that time, right? They're just meant to kind of discuss the time, spread it across the system and no one else can have the power to increment the time, and that was very important to us.

[00:47:52] JM: I'd like to talk some about your work, because you have a pretty unique role doing distributed systems product management essentially. Describe how user experience fits into the design of the database.

[00:48:10] AC: Yeah, definitely. So most product managers, I say, would have a UI, a user interface, that their users can interact with. With that user interface, you can do things like A-B testing. You can kind of understand how your users react to changes in a rolling fashion, right? Kind of task to be iterative.

Now, in MongoDB, we don't have a UI. We don't have a thing that we can easily iterate on. Once we put an API out there, we have to support that API. Otherwise, we're going to break applications that adapted the API, right? We can just change in API out from under somebody.

So we could change in API, but then we'll just add a new version of it and continue supporting the old version. Then if you add to many versions of the same API, you're getting a bloated product. You're supporting every version of the API that anyone has ever used, which can be very challenging that tradeoffs of like sticking with an API that you've designed, because otherwise you're going to have so many different versions of the API and continue supporting the old one anyway.

So they're a bunch of interesting challenges for working on a product where the user experience is really defined by the API and the documentation, right? Those are first-class citizens. Those are our UI. That is how users engage with the product.

Now, being a product manager in not world, first, is very technical, right? You have to be able to empathize with the user, and the users are very technical. So you have to understand those problems. But it also creates a lot of necessity to go and analyze usage data, but also talk to people. Understand the whys. Like what they enjoy about certain APIs and bring that back to the team and make sure that we're looking at all of these APIs from a user-centric perspective, right? There is a need for product management here even though there's not a direct UI element.

[00:50:13] JM: What was the process for rolling out the update to MongoDB? So when you implemented causal consistency, you have to roll this out or make it an optional update. What's the process for getting that update out?

[00:50:31] AC: Yes. So causal consistency was included in our 3.6 major version. So MongoDB 3.6 has the causal consistency on the server side. Now, this feature actually also required drivers changes, which are basically our Python driver, Go driver, right? They all had to implement the API to leverage the server feature. So we kind of had to coordinate those releases across both the languages and the server corresponding with our MongoDB 3.6 GA.

Now, anyone who is hosting MongoDB themselves is completely empowered to upgrade at their own leisure. They can upgrade whenever they want. Some people are still on 1.6. Some people are on 3.2. Some people are on our newer versions, which 4.2, which just GA's last month, and then 4.0, which a year ago at this point. So there is a wide range of different versions out there, and as a product manager for a product that you can host yourself, we have very little control over when people decide to upgrade.

Now, people only want to go through the pain of upgrading something when they feel motivated and the feature set is strong enough that they can start taking advantage or really make their lives easier or other ways by upgrading. So it's really about kind of educating what's in the new releases so that people feel like they want to take advantage of the cool stuff we're building.

[00:51:52] JM: You mentioned going to the database conference and presenting this paper. Tell me about the state-of-the-art in distributed database development. What are the new were conversations or the newer algorithms? What's the hot topic in distributed systems database design these days?

[00:52:16] AC: There're tons of great stuff right now. I really like the conversations around TLA+. Not that that's new or anything, but that's a specification where you can kind of prove the behavior of your system by writing like a system. That's been really great. We've been doing that across many features and starting to kind of play around with this in different areas in MongoDB, and we're really excited about that. There's also this concept. I think it's Peter Baelish paper called Probabilistic Staleness.

Again, I think it's like seven years old at this point, but we're just starting to see some implementations in the wile, and that's pretty exciting in like in production systems, which is great. It does take time to move from academia to products, and it's cool to kind of track those concept over time.

[00:53:07] JM: Well, that's for sure. I mean, it seems like a recurring feature in Leslie Lamport's inventions, where it came out with Paxos and then it took like – I don't know, 5 or 10 years or something for it to actually be recognized by the software engineering community. Then kind of the same thing happen with TLA+, where he invented this thing for verifying your distributed system and then just nobody used it for a long time. Then finally somebody started using it at Amazon and then more and more people started using it. It sounds like you're using it now.

[00:53:43] AC: Yup. The world works in weird ways.

[00:53:48] JM: So TLA+, that's for verifying the specific distributed systems characteristics of a system work the way you think they do?

[00:54:00] AC: Yes, exactly.

[00:54:02] JM: What's an example of how you've used that?

[00:54:06] AC: So one of the ways that we're using it right now is to verify offline application. So that when secondaries are replicating the operation log or oplog, that we are confirming that they're replicating in the way that we wrote it, of the way that we expect. So kind of making that more provable and kind of building out these proofs across kind of smaller components in our distributed system and like proving the behavior at a granular level across the board.

[00:54:33] JM: Okay, last question. How will MongoDB change in the next five years?

[00:54:38] AC: Oh! In so many ways, and some of those are going to be hard to predict sitting here today what MongoDB is going to look like in five years. I think that we will continue to try and solve problems that people are having to hack away at in application development today, right? So our goal is to take away all of the things that are not beneficial to having the team work on away from the apt teams and into the system as a first-class citizen, right? We don't want you to have to hack around targeting nodes yourself. We want to build something that you can write a query and then role route to the right nodes, right? That's conceptually what we want to continue doing.

One of the interesting things that we're looking at now is around how to minimize your P99s, like your P95 latency, read latencies, and make it more predictable that you don't get stuck with a bad node, a slow performing node. Today people do that by just targeting multiple nodes in their application and servicing the results from the fastest note, right? This is a concept called hedge reads. We've seenthis in Google. We've seen it an Amazon. They've released some papers about it. We want to make that first-class citizen, right? We want to bring stuff like that into the server. That's just an idea, right? But it's fun, right? These are fun problems that we can bring and ultimately make it so that developers don't have to build it themselves.

[00:56:07] JM: Aly Cabral, thank you for coming on Software Engineering Daily.

[00:56:09] AC: Thank you so much for having me.

[END OF INTERVIEW]

[00:56:20] JM: Software Engineering Daily reaches 30,000 engineers every week day, and 250,000 engineers every month. If you'd like to sponsor Software Engineering Daily, send us an email, sponsor@softwareengineeringdaily.com. Reaching developers and technical audiences is not easy, and we've spent the last four years developing a trusted relationship with our audience. We don't accept every advertiser, because we work closely with our advertisers and we make sure that the product is something that can be useful to our listeners. Developers are always looking to save time and money, and developers are happy to purchase products that fulfill this goal.

You can send us an email at sponsor@softwareengineering.com even if you're just curious about sponsorships. You can feel free to send us an email with a variety of sponsorship packages and options.

Thanks for listening.

[END]