EPISODE 686

[INTRODUCTION]

[00:00:00] JM: When a database gets large, it can start to perform poorly. This can manifest in slow query speed. You can speed up a query by defining an index, which is a data structure that allows for faster access to the data that is being indexed. As a consequence, whenever you update that database, you will now need to update the index with that new piece of data. The more you index your data, the faster the access time. In order to have more indexes, you must pay a right penalty in order to maintain consistency around that data since the indexes need to be updated with each new entry, and this illustrates just one simple trade-off that a developer can make within a database deployment.

The question of tradeoffs is why there are so many different databases in the world. Obviously, these different databases can mostly fulfill our basic needs of storing and retrieving data. So why do we need SQL databases like PostgreS? Why do we need to document databases like MongoDB and key-value systems like Cassandra and search systems like Elasticsearch? It's because each of these systems are optimized for different sets of tradeoffs, and tradeoffs can affect the speed of a read, the speed of a write, the user experience, the consistency of data and the cost of running the database. Learning about these different database features and trade-offs can help you understand how to evaluate which database to choose and how to optimize and assess the performance of a database that you've already deployed to see if maybe you want to tweak it, or maybe you need to switch to an entire database altogether.

Andrew Davidson is the lead product manager of MongoDB Atlas. Andrew joins the show to talk about how database performance can degrade when a database gets large and how to measure and optimize the performance of a critical database. Andrew explores the range of distributed systems cases, from a single node database to a multi-geographic distribution of nodes around the world, and he describes how the configuration of the database in the cloud can help or hurt the application that the database is serving. Full disclosure; MongoDB, where Andrew works, is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:02:31] JM: I have learned a ton from QCon. QCon San Francisco takes place this year, November 5th through 9th, 2018, and I will be going for my fourth year in a row. I always love going and seeing the talks, and in between the talks I hang out and eat some mixed nuts, chat with other engineering leadership about the latest talks and stuff that they're seeing, the 50 different stream processing systems that we're seeing, the different databases we're seeing, and QCon is a place where senior software developers and team leaders and managers and senior leaders, they all gather together, and you have fantastic conversations. You have fantastic presentations, and it's extremely high quality. Its technical. A lot of it is also cultural and about management, and you can get $100 by using the code SED100.

QCon is a great learning experience. The presentations this year include 18 editorial tracks with more than 140 speakers from places like Uber, Google, Dropbox, Slack, Twitter. They are curated high quality talks, and you can get $100 off if you use code SED100 at checkout for your ticket. Again, QCon San Francisco takes place November 5th through 9th, 2018, and the conference is the 4th through 7th, November 8th through 9th are the workshops. You can go to qconsf.com to find out more.

Thank you to QCon. If you are going to buy a ticket, please use code SED100, and we really appreciate the sponsorship of QCon.

[INTERVIEW]

[00:04:28] JM: Andrew Davidson, you are the lead product manager at MongoDB Atlas. Welcome to Software Engineering Daily.

[00:04:34] AD: Thanks so much, Jeff. Great to be here.

[00:04:36] JM: I want to talk today about database performance with you, and the first part of the topic that I'd like to explore is why there are different databases? Because there must be some different tradeoffs between these different databases, but to some degree they are giving us the same thing. We ask them for data. They give us data back. Why are there so many databases?

[00:05:00] AD: It's a great question. I mean, there are so many databases basically because the database and the data are the heart and soul of the application and they're just fundamentally critical to building great application experiences no matter what we're building today. I think just as we're sort of seeing this new trend where people are actually starting to build custom semiconductor level hardware for various applications, where just something you have to do if you want to optimize a new IoT devise, you go one layer up in the stack.

If you're building an application and it doesn't have the right impedance match to its data store, it can really fundamentally slow down what that application could do. It's not necessarily really all about performance. I think a lot of times folks talk about performance or benchmarks and such, but really the most important thing is often how basically when you're building the application and iterating on it, it's a question of, for the developers building that application, can they continue to iterate and achieve their goals at the beginning, the middle and also at scale as they – Or maintenance mode for years ahead? All of those are critical things that, fundamentally, are set based on this impedance match between their application or database.

A lot of folks have gone and built new databases because they had some novel application they're trying to build. In fact, MongoDB came out originally because the founders of MongoDB were trying to build really platform as a service and they needed a kind of a database to back it, and then eventually realized that the database itself was such a big project and caught on, they should continue with that. I think that's happened to a lot of folks.

I read a fun Hacker News comment from someone once who is basically saying if a project involves inside of it a database project, that it is just a database project essentially. I think it was a riff on someone who said if a project involves building a space launch vehicle, it's just a space launch vehicle project. In other words databases are extremely big things, and so folks often will end up building them for custom use cases.

[00:06:51] JM: Right. There are different tradeoffs that different databases can make, and these tradeoffs can exist on the axis of performance, like the speed of a write or the speed of a read, but also on the developer experience or some UI layer that you build to make it easy for people

to interface with the database from a web browser, for example. You can also make tradeoffs on the axis of cost. What are some of the different tradeoffs that a database can make?

[00:07:24] AD: Let's just start with the sort of the elephant in the room when you're talking about MongoDB in particular, is fundamentally it's this very different data model from our traditional table-based relational databases. In other words, with a relational database, which came out of the 1970s world, in which very different time. At that time the cost bottleneck really was all about storage. In other words, if you could keep storage costs low, that was of incredible value. You could throw as many people as you want at the problem, because your storage was the key bottleneck. That was the 70s. This idea of having a bunch of tables linked with keys and having your objects that's sort of mapped across them all made a lot of sense when storage was all that mattered to keep down.

With MongoDB, what happens is all of a sudden this concept of a particular object in an application code being spread across a bunch of tables goes away. Typically, an object, instead of it requiring a join across a bunch of tables, an object is simply described the same way in your job, or your Python code, as it would actually be in the database as a document similar to JSON and Structure. This, of course, obviously uses more storage, because you're de-normalizing data across different environments, but of course today storage is really cheap, and that's not generally the problem. The problem today, the bottleneck today, is developers time. Basically, the ability to quickly understand what's going on and make changes, change their application to build new capabilities.

When you don't have to think about these complex joins across lots and lots of tables, you get a lot of benefits from a developer experience perspective. If you look at a lot of code, typically a developer writing code with a relational database will have an object relational map, an ORM layer, and essentially that later is something – Some people love it. Some people hate it. In any event, that's kind of a key layer where you get this bizarre impedance mismatch when you're going to a bunch of tables underneath. That letter effectively gets replaced by the database when you move to MongoDB.

[00:09:21] JM: So there's also the ability to move between different databases for different applications. So, for example, Mongo is often used as a transactional layer, because developers

can write their frontend application code against that Mongo database, because the structure of the document model is, like you said, reflective of the – The way that I want to think about my application level service is the way I want to think about my customer-facing services. I want to make a write to a user document or a post document. I want to create an order, and I want to think of these things as JSON document models because that's where most of the interactions – That's how objects are structured on the frontend, and so it's how you want object structured in your transactional layer. But there's also the option to move that data from the transactional layer into other databases in order to perform jobs that are more reflective of certain backend processes.

So if I'm writing a frontend service, maybe it makes sense for me to be in the paradigm of updating document models. But that doesn't mean that I can't have ETL jobs, these extract transform and load jobs that transform that transactional data into a MySQL database, or a data lake that's in parquet files that are column oriented. You can have your operational data map into other data formats in order to write backend applications that might need to grab everything in a column.

Could you talk about that in a little more detail? When do people want to do ETL jobs from one data system to another and why do they need to have these ETL job, and why does that relate to database performance?

[00:11:34] AD: I think I'll start off by saying nobody wants to do ETL, and that it tends to be sort of, typically, the things that folks will find are further removed from where their businesses, where their customer is, and it's kind of stuff that they wish they didn't have to do, but nevertheless have to do, and they have to do it a lot of the time, because they have critical stakeholders. Whether it's business analysts, or product managers, or an executive dashboard who just need to be able to understand what the application data is telling them so that the company or the application team can make better decisions about where to go and track what's going on.

So I think they use ETL because the pain point is though stakeholders can't easily learn those things if they don't or traditionally have not been able to. To your point, yes, they go to data stores that are optimized for queries on a particular field, like a column store, and what that

means is sort of at the I/O level, with a column store, you're pulling out from every page from disk. You're essentially pulling out a bunch of values from a bunch of different rows or MongoDB nomenclature, a bunch of different values from a bunch of different documents altogether without getting all the other columns or fields from those same documents or rows, which is what a row store, or which is sort of the MongoDB model, more of a B-tree model.

Yeah, columnar storage going truly offline into object storage, like you said, maybe in parquet or CSV. These are all things that a lot of folks do so that they can more easily service those other kinds of stakeholders. I will point out that what a lot of folks will do with MongoDB as an alternative to doing ETL is you can use read replicas for workload isolation, and that way you're not impacting the performance of your main cluster, but those replicas are already there for high-availability anyway, and you could have sort of offline analytical aggregation type queries against them. You can also do a – The other key thing is a lot of these business analysts, they might query the data with just a different query language. They that might really want to use SQL. SQL is not ideal for developers, because it's that impedance mismatch to their applications, but a business analyst worldview where they to declaratively describe what they want to pull back, a lot of them love SQL.

So the ability to kind of ETL into a SQL native data storage, tt can be compelling to them. MongoDB does have a BI connector which will convert MongoDB to SQL, but it doesn't have, for example, the columnar performance advantages. So it's a question of do you optimize for performance from the data store, but then you're having to do all these ETL type work, or do you sort of perhaps optimize for reducing things like the ETL plumbing but leverage converters that make different data stores look like other kinds of data stores?

[00:14:19] JM: You're saying that in some cases, ETL works. But in other cases, you may want to do something else.

[00:14:26] AD: Yeah. I mean, basically, ETL is a cumbersome thing to do. You could do ETL if you really want to move into a data store that will be like the best performance for your data analyst. But if you're willing to have your data analyst use okay performance and not have to use, for example, do all the ETL work, then you can actually just do things like the MongoDB BI connector which will speaks SQL for an analyst to the database to MongoDB natively, and you

could do that against read replicas so you have workload isolation, not hammering your production database.

[00:14:56] JM: What is the point at which the approach without ETL or one of the approaches without ETL. So either the read isolated replica or maybe just having the business analyst accessing the operational database, because you don't have that much data, or the workload is just not too intensive. But what's the point where things start to fall over and you do need to do ETL?

[00:15:23] AD: I think it all really depends. I mean, every company is different. Like some folks, for example, want to move to HDFS, because they already have a big investment in a large Hadoop cluster. Although I think I'd be curious to hear what you think, Jeff. But over the last two years, I think we're seeing a bit of a shift away from excitement over HDFS and Hadoop towards, really, it's more about compute at the Spark tier without it being all about HDFS. You could do things like Spark natively against MongoDB as well against those read replicas.

I think it really just depends. I mean, if you're trying to do massive scale machine learning, you don't necessarily want to do that off of your read replicas of your operational data store, but if you're having your data analyst do the key weekly reporting on the context of your application, then that's something very different and something that certainly you might not care so much about having their queries run a little bit longer than they would if they're in an optimized data store.

[00:16:18] JM: Yeah, that's consistent with what I've heard from other people. So there are some cases where you can speed up your queries by doing things like database normalization or indexing. Can you explain what these approaches are? Because I want to get into a discussion of some tactics involving database performance and database optimization. So maybe could define these terms for us, indexing and database normalization.

[00:16:52] AD: Frankly, indexing is really the most important kind of performance 101 aspect of MongoDB to be aware of. Really, what indexing is all about is, out of a gate, a default MongoDB collection will have one particular field that is indexed, _ID it's called. It's kind of like the always there field in MongoDB.

What this means is if you're running a query against a MongoDB collection, by default you're going to have to go through every single document in that collection to figure out whether the results of your query are hit, unless you're querying on _ID. If you have to go through every document in the collection in order to provide the results, that's what we call collection scanner, or traditionally a table scan, that basically is very slow. It doesn't always feel slow when you have a small data size and you kind of could ignore it for a while. But as you get a bigger dataset, essentially, you're not using a binary search tree. You're actually going through every item in order. So the speed will just scale linearly with the size of the dataset.

Now, indexes come in where you can actually, in MongoDB, decide to add secondary indexes to any field in the document, and those can also be nested fields and it can actually be arrays as well. You can index arrays, and you can also create a lot of kind of more advanced indexes. You can do geospatial indexes, full-text search indexes. Probably the most important is you can do compound indexes. You can have an index that involves multiple fields, which allows you to do things like a query that will involve multiple fields, like maybe a type of clothing and also size. If I indexed on both type of clothing and size, I could very quickly provide all kinds of results for someone for whom. I know they're look at pants. I know what size they're in.

So this ability that uses advanced indexes, they allow you to not have to do these table scans where time will grow linearly with data size, and instead essentially have more of the search tree or [inaudible 00:18:47] time dynamic, which essentially feels very snappy to the end-user. So indexing is just critical.

Now, normalization versus de-normalization, this basically is all about do you store data – So MongoDB is all about the de-normalization for the most part. You're storing data about an object in the document where that object is stored. So, for example, if we're storing a collection about clothing and we've got t-shirts, we're going to describe the colors that the t-shirt has right there on the t-shirt document. So if we have a blue t-shirt that's available in one t-shirt brand, we might also have a blue t-shirt that's available in another t-shirt brand, and that's all just fine. They're just attributes of the t-shirts.

The normalized model would be to sort of have a separate table where we describe the potential colors the T-shirt could have, blue, red, etc., and we would actually not store this concept of blue right there in our object right with our t-shirt, but actually look it up through a join. Kind of going back to the beginning, that made sense where it was so critical that essentially we not store blue in multiple documents, because storage was so scarce then. But nowadays the idea of having to do a join just to find out what t-shirt colors are available for t-shirt is just cumbersome.

So de-normalization is kind of the new way to go for developers where it's easier for them to understand what's going on. Where de-normalization gets complex is if you have different kinds of data that are related in different documents and you need to be able to transactionally updated them together. In other words, if I wanted to have the number of t-shirts available in one collection for a certain t-shirt brand, but also have sort of another collection, which is the number of – Like someone's check out cart, their orders, and if I wanted to have myself add a t-shirt to my cart and also ensure that no one else could therefore get the t-shirt into their cart, and therefore kind of decrement the t-shirt counter, I would have to be changing a value involving multiple documents, and that's something that traditionally was very difficult with MongoDB. MongoDB did not have multi-document ACID transactions for the first, call it, 10 years of its database life.

One thing that came with MongoDB 4.0, which was released a few months ago in June, was for the first time of MongoDB, multi-document ACID transactions. So that canonical example of the decrement, the t-shirt counter so I can check it out and not have someone else, is now so that it can be easily done in one's code.

[SPONSOR MESSAGE]

[00:21:26] JM: Accenture is hiring software engineers and architects skilled in modern cloud native tech. If you're looking for a job, check out open opportunities at accenture.com/cloud-native-careers. That's accenture.com/cloud-native-careers.

Working with over 90% of the Fortune 100 companies, Accenture is creating innovative, cutting-edge applications for the cloud, and they are the number one integrator for Amazon Web

Services, Microsoft Azure, Google Cloud Platform and more. Accenture innovators come from diverse backgrounds and cultures and they work together to solve client's most challenging problems.

Accenture is committed to developing talent, empowering you with leading edge technology and providing exceptional support to shape your future and work with a global collective that's shaping the future of technology.

Accenture's technology academy, established with MIT, is just one example of how they will equip you with the latest tech skills. That's why they've been recognized on Fortune 100's best companies to work for list for 10 consecutive years.

Grow your career while continuously learning, creating and applying new cloud solutions now. Apply for a job at Accenture today by going to accenture.com/cloud-native-careers. That's accenture.com/cloud-native-careers.

[INTERVIEW CONTINUED]

[00:23:05] JM: Let's dive deeper into indexing.

[00:23:07] AD: Sure.

[00:23:07] JM: My sense is that indexing gives you faster speed in exchange for maintaining an extra data store. So there's a slight tradeoff between speed and more storage and just an increase in the complexity of your database, because you're adding in this new data structure.

[00:23:30] AD: Yeah. Actually, it's really the key – You're hitting the nail on the head. The key thing, the key tradeoff with indexes is you start paying a write penalty. MongoDB is a strongly consistent database. Meaning, if I'm running a query and it's served by an index, the index will always serve a query for the latest view of the data. The index will never store like a view of the data that hasn't yet become updated yet. It's not eventually consistent.

As a result, indexes are on a critical path to writes. Yeah, you're hitting the nail on the head. It's not uncommon where you see someone who has like 20 fields in their documents and they'll have like all 20 fields indexed, and that is something that definitely could be that kind of rookie mistake where you might think that makes sense, but then every time you're doing a write or an update, you're potentially having to not just update one place on disk, but you're effectively updating 20 places on disk. There's amplification of the write, the I/O required, which absolutely slows things down.

[00:24:31] JM: Why is that? Can't you update the index asynchronously from updating the actual field that you're writing to?

[00:24:40] AD: So you could do that, but then what's happening is in order to use the index and provide a consistent result, you have to confirm whether or not the index is indeed the latest view of the data or not. So you're paying that penalty somewhere where you either have to go check with the document's current state to see if the indexed view is the current state. The alternative is to actually say it's not uncommon for people to say, "I don't need consistency, and I'm happy to have a search engine." A lot of folks will use search engines that are fundamentally eventually consistent, because you're pushing your data at your operational data [inaudible 00:25:12] search engine, and a lot of the times folks will say, "That's okay. For the most part, I'm fine with that." It's just a different tradeoff. But inside the core operational database, at this time anyway, MongoDB does not offer that eventually consistent indexing model.

[00:25:26] JM: So the consistency tradeoff here is if there's a field that I write to that has an index defined over it, and as I write to that document, if somebody does a lookup for that same object by that index and the lookup in that index is not locked, then they might get a read of data that is now stale, because my write has not finished. So they're going to read stale – The question is they're either going to read stale data or they're going to have to wait until both the document itself and the index that tracks all those documents that are associated with that index gets fully updated. So you have a tradeoff in consistency versus time there.

[00:26:23] AD: Yeah. You hit the nail on the head. This is of course why if you kind of look way down in the stack, the very bottom of the database stack is typically a storage engine. This is why storage engines are such just critical part of what makes the database. If you look back at

MongoDB's history, probably for the first, I'm going to call it five or even six years of MongoDB, the built-in storage engine, which was called MMAPv1, it was probably not the strongest part of the database. I think nobody would think it was.

MongoDB was built with great high-availability, distributed systems capabilities, replication and sharding, but the storage engine in those first few years of MongoDB, fundamentally, a lot of people who tried MongoDB in those days, they ran into all kinds of locking contention issues, because it's so critical, especially when you start talking about like what we're talking about here, which is document level writes and ensuring that you're not locking for longer than you need to during the context of the writer or the write to your index.

MongoDB brought in this new storage engine about five years ago called WiredTiger. It's been the default since 3.2 of MongoDB. A couple of major releases back. Basically, WiredTiger was built by this team that had kind of legends in the storage engine space, had built some of significant storage engine offerings elsewhere before they built this offering. MongoDB acquired WiredTiger, made that the core storage engine of the database. All of a sudden you get that true document level concurrency, which I would encourage if anyone is listening today who used MongoDB back in those early days where you ran into those locking issues, really, everything we're talking about now, it's all about true concurrency at the document level, which is really what you want. I mean, object level concurrency.

[00:28:09] JM: Let's revisit normalization. What are the performance tradeoffs of normalization?

[00:28:16] AD: Well, with normalization, you're essentially having to look at multiple places to – Essentially, if you want to know, going back to the t-shirt example. If I quickly want to know everything I need to know about my t-shirt, if I can just look it to one place on disk very quickly just to pull back my document record, then I have to actually jump to a bunch of other places to get the detail about my t-shirt, and that requires us to do kind of similar to the indexing, but the reverse, where adding lots of indexes will force us to do lots of different writes at the disk level where we're doing a single document write.

With normalization, on the read path, we're having to do lots and lots of different reads from lots of different places on disk effectively in order to pull everything together, which is you don't want

to – That could be frustrating if an object you wanted back as quickly as possible, because you care about everything in that object, or a subset of the portions of that object that matter to you, not having to have that spread across a bunch of different contacts is an advantage.

It's particularly an advantage – Normalization could be particularly complex in a distributed or sharded cluster where you ideally would not want to have to join data, normalize data across different shards or different nodes, because then you're talking about network hops as well to bring the data together. Now, of course, most of the relational databases don't have a kind of distributed paradigm to them. They're typically stand alone or more application to your sharding with a high-availability built in.

So that often doesn't necessarily come up, but with like scale out databases like MongoDB where sharding is a first-class citizen, you want to think about whether or not you're having to join across multiple nodes or if you can keep things within an object, then everything is faster.

[00:30:02] JM: So if I'm working with a database and my database starts to go slow. I'm looking at it. What kinds of introspection can I do to figure out what's going slow and how can I use these primitives, these speed up primitives, like indexing, or normalization, to make my performance better?

[00:30:25] AD: Sure. A lot of folks who come over from a relational background and maybe they're using MySQL and they move over to MongoDB, it's not uncommon to bring the schema of modeling ideas from relational to MongoDB, and all of a sudden, where they might've had end tables in MySQL, they might use end collections in MongoDB, and this is like a big red flag to begin with. Because what is happening is they're using a database that is really built primarily for de-normalize, getting the description of the object together in place and indexing it therein. Instead, you are doing a bunch of joins where you don't have to, and it becomes a bigger no-no when you're talking about a distributed system.

From first principles, it's really important to think about your schema, and basically with MongoDB, -be describe objects, business entities as an object in your code and store that the same way inside your collections. When you do need to do a join in MongoDB, which is not

something that happens, but far less than in relational, because it's not on every object view that you have to do the join, there's something called dollar look up that lets you do it.

Assuming your schema fundamentally reflects what your objects are, and that's so important for MongoDB. Then from there, you want to index on, really, what are the most important queries that you need to deliver great SLAs for? That's different for everyone. I think it's fair to say that the queries that are happening the most frequently that are on the critical path to your end user experiences, those are the ones that you probably want to index and you want to look at what you're querying for and simply index on that subset of fields. Make a compound index, for example.

[00:32:01] JM: So now I'd like to move into talking about distributed systems and add another axis of complexity into our discussion. First, explain what a primary and a replica database are and some typical setups for a database that involves a primary and a replica.

[00:32:21] AD: Sure. MongoDB uses a Raft-like consensus algorithm, which is pretty much the standard today. The great thing about this consensus algorithm is essentially what you do is you have at least three replicas. That's the idea. The reason you want three is that this allows you to do auto failover. You can't actually do auto failover with just two, and I'll explain why. If you've got just two, then essentially – And this kind of goes back to the CAP theorem; consistency, availability, partition, tolerance.

Basically, if you have just to nodes and once is a replica of the other, there's no way for either of the two nodes to be certain that it should or should not continue to take write, because it doesn't know – Basically, if there's a partition between the two, each one will believe the other is down. So they could either both take writes, which can lead to a consistency quagmire, where you have different potential application instances writing to different perceived primaries at the same time. That's a kind of a nightmare scenario, because there's all kinds of conflicts that can arise, or you can have both nodes, both replicas sort of say, "I can't see the other one. Therefore, I can't take writes right now." Then all of a sudden you lose write availability in the event of a partition.

The beautiful thing about three, and a three is the standard with MongoDB. If you have a replica set which is sort of the simplest clusters in MongoDB, it's always going to be three. There's options to do more than that, the three is kind of the default, or if you have a sharded cluster, each shard is just a replica set of three, again. So it's the same availability, but it just scaled out.

The beautiful thing about three is if you have a partition, the idea is that you would need to actually partition between all three to lose that write availability, and that's very unlikely to happen. The key is that any individual node being down when there's 3, or any individual node that appears to be down when there's three to two of the three, the two of three can continue to form a majority consensus, and amongst the two of them elect a primary where writes will go in. This allows you to basically get the high-availability required today where the cost is, one, you need to have three data nodes. That's a key cost. Two, you need to be – This model, as far as the CAP theorem goes, essentially, you're saying, "I'm willing to sacrifice availability." Meaning, if I lose two of my three nodes, if I lose the majority, then I will stop taking writes.

But this model and to MongoDB model, and I think it's kind of the emerging standard. This model says, "It's so unlikely that I'll lose a majority. I'd rather optimize for not taking writes when the majority is lost," then instead optimize for taking writes in multiple locations where consistency becomes such a problem. MongoDB has always wanted to optimize for consistency. With distributed system for availability and modern infrastructure, particularly in the cloud, like MongoDB Atlas, where I'm the product manager for, where we can automatically spread the cluster nodes across availability zones, which are essentially distinct data centers on the backend of the cloud providers, or you could even go across cloud regions, so three or more cloud regions. The kinds of availability you can get today with three and knowing that you'll have to have a majority up, you can get easily four nines with that. As a result, optimizing for consistency makes a lot of sense.

[00:35:44] JM: We explored a trade-off of consistency versus performance a little bit earlier when we're talking about the question of indexing and the fact that when you define an index, the tradeoff, one tradeoff that you're making, is that the index will have to be updated in a consistent fashion with a write. If you have a concurrent read, then that reed is going to have to wait around. Now, that's a single node example of this consistency question, this performance question. With the introduction of multiple nodes, we have a variety. We have a raft of new

questions around consistency and performance that we can explore. Explain why the multi-node setup can affect performance and what are the different axes of performance improvements that we can explore in the context of multiple nodes?

[00:36:52] AD: Sure. There's really two key dimensions when we're talking multi-node distributed MongoDB clusters. There's others the replication where you have the same data on multiple nodes, and that's for high-availability and for workload isolation we talked about the beginning where you might want to have certain classes of queries target a read replica so that they don't disrupt the primary where rights are going. But then you also have separately sharding, which is scale out. The ability to say, "I'm going to take multiple replica sets and distribute my collection across them."

So this allows us to do long-term linear scale out into, really, any level of scale. If you want to go into multiple terabytes, it makes sense to start sharding, because it's just – You have a lot of concurrency benefits not having to have multiple terabytes on one particular note node.

So when it comes to consistency, it's a great question. When you're reading from a replica, like we mentioned, you might have your business analyst do, or you might just have – Or another example might be you might have read replicas for your – Like in MongoDB Atlas, we make it easy to define global clusters around the world, and you might want to have a read replica from your U.S. data in Europe or in Singapore to enable low latency reads from those other parts of the world. There's a consistency tradeoff there in so far as you don't want to read from a secondary that may be really behind the primary or maybe you're willing to. You just have to be aware of whether or not you need to know that when you're reading from a replica, that data may or may not be the latest data.

There are some knobs to consider MongoDB exposes, like you could do something called read concern majority, which basically states that you're reading something that has gone to a majority of the nodes in the cluster, which guarantees that it will never be rolled back. A rollback is what can occur when you have an election like you lose a node in your cluster, and that was the primary, and you have to elect a new primary, and that happens before the rights have gone to the majority of the nodes in the cluster.

So you can use this read concern majority to know that what you're reading will never be rolled back, but the only way to read to be absolutely certain that what you're reading is the latest that the primary has would be to read directly to the primary. So various tradeoffs to consider.

[00:38:59] JM: The question of multi-node in a single data center or even in a single geographical region with multiple data centers can be different than a question of global consistency, where you might have one data center in the U.S. and one data center in Asia, because there can be significant latency and more questions of consistent network bandwidth between those two data centers or between those two geographic regions. How does the geographical, the global nature of a – If you want global consistency, or the questions you can explore as to whether or not we need global consistency and the penalties that you get for trying to maintain global consistency. How does the question of global consistency? If we're talking global in the sense of geographically global, like in multiple continents, how does that affect – If we're talking about, "Oh, if some global application where we might need consistency, like we're building Gmail, we probably need some global consistency."

[00:40:12] AD: There's a couple of options to consider. I mean, the model we've been talking about so far where there's essentially a primary – Let's take a step back from the data center. Let's imagine that we kind of describe the cloud regions that we're thinking about deploying into and let's say we describe the preferred region. This is something you could do in Atlas, in MongoDB Atlas, for example, on AWS, Google Cloud or Azure, you choose your region in Atlas, you choose a size of the cluster, but you can also have it be multi-region.

Now, let's say you choose a preferred region, U.S. East, for example. You can have those read replica regions elsewhere, but fundamentally, all of those writes by default are to go into the U.S. East region. So there're two things. It's not just consistency. Insofar as those read replicas might fall behind the primary, I would argue that actually the bigger problem will probably not really be one of consistency globally, but instead that, frankly, your users who are over there in Singapore or in Europe, their write, that the latency for their writes are going to be painfully slow, because they're having to wait for a round trip all the way to the U.S. and back.

Really, write latency is the key. Therefore, the question is; can you have a cluster that puts the primary, so to speak, associated with the data near to where the end-users of that data are and

do that in different parts the world? We have something called global clusters in MongoDB Atlas, which use sort of a very opinionated version of MongoDB's zone sharding to do just that. Essentially what you do is you take a global sharded cluster and you essentially are choosing which subsets of shards will be homed in which parts of the world. So you could have a U.S. shard, a Europe shard and an Asia shard, and what happens is you can have U.S. users write and read locally to the U.S. shard. European users write and read locally to the Europe shard, and Asia users write and read locally to the Asia shard.

So for the users, they get what they want, which is the in-region consistent and low latency experience for reads and writes. But then you still have the ability to do global aggregations across all of the data because it's a single cluster. So what you don't get from what I just described is the ability to have read replicas from each of those zones in the other zone, but you can do that. So in Atlas, you could opt in to that so I could have a read replica from our U.S. data in Europe and in Singapore, read replica from my Europe data and U.S. and Singapore data could have rad replica in Europe and the U.S. When we do this, we can get in-region low latency reads for the global data with the compromise being that the global data from the other regions will be from a read replica, and therefore could be a bit stale. All things to consider, and if you wanted to get up to date region query from the other regions, you just fundamentally have to do a round trip.

[SPONSOR MESSAGE]

[00:43:12] JM: DigitalOcean is a reliable, easy to use cloud provider. I've used DigitalOcean for years whenever I want to get an application off the ground quickly, and I've always loved the focus on user experience, the great documentation and the simple user interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU optimized droplets, perfect for highly active frontend servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances

have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make a hundred dollars go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure, and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.

[END OF INTERVIEW]

[00:45:19] JM: We've now talked about theoretical concerns at the single node level and at the distributed systems level, and the bottlenecks that a developer might have as their database scales up. I want to talk now about product development of building databases, because you work at MongoDB. You lead product management at MongoDB Atlas. I'd like talk about MongoDB Atlas as a case study in building a database system that satisfies some of the challenges of the developer that's working on a large-scale database. So what is MongoDB Atlas and what you do as the product manager on it?

[00:46:10] AD: Yeah, sure. MongoDB Atlas is an elastic database as a service for MongoDB. It's available on the big three public cloud providers; AWS, Microsoft Azure and Google Cloud. But MongoDB Atlas is the kind of thing where you sign up on our website. In other words, it's our service, but you choose exactly which cloud region to deploy your cluster into and then you can connect locally from your application infrastructure in that same region to get in-region latencies.

Atlas has all these capabilities, like you can scale your cluster up and down with no downtime. You get all the operational requirements you could ever want, for monitoring and alerting, to backups, and you could do a lot of those key workflows, likes restoring or cloning a backup from production into a staging environment so you can test next week's production code against last

week's production data. All those kinds of things are easy to do throughout Atlas, because everything you could do in the UI, you can also do through the API. In another words, it's very much built in the infrastructure as code paradigm to be very developer-centered.

From a product management perspective, it's very exciting, but it's also fundamentally a challenge, because MongoDB Atlas is a general-purpose platform as MongoDB is. We think about first-class customer experiences ranging from the solo developer who may be a student trying a new side project, and we in fact have a free forever free tier symbol MongoDB cluster with half a gig storage. Anyone can deploy that and sign up in Atlas, get it today, and never have to worry about paying for it. That's there to ensure that folks just always can try new ideas on this. It's like an alternative to the laptop but running it in the cloud.

We think about that solo developer who's starting with a new idea or maybe learning about MongoDB for the first time, but we also think a lot about an application team that's seriously in production. They're thinking about what they need to do to scale and basically what their journey looks like to have an optimized set of workflows so that they can do as little of the plumbing as possible and focus fundamentally on their application, on their end customers as much as possible.

Then we also think about kind of what you might call the more traditional enterprise customer who will have a lot of the traits of that second category I mentioned, but also will require security best practices be fully auditable and the compliance capabilities and really have centralized controls that someone who's more in the, "I'm a smaller startup. I'm not thinking as much about." The bigger enterprise requires a lot more of those capabilities.

We really think about optimizing for all three of those categories. We think all three are just critical to the long-term for MongoDB Atlas, because we think that someone who starts in one can very easily grow into the next one, and our goal is to really be a ubiquitous data layer that's easy to access for developers, really, at any level of adoption.

[00:48:59] JM: I think of this category of products as database as a service, and it relates to the platform as a service model. So if you go back to the early days, I guess Mongo even came out of a platform as a service approach, as you mentioned earlier. They were trying to build a

platform as a service. They ended up building a database, and now I guess they've kind of come full circle, because database as a service.

So when I think about my first time deploying to AWS, remember doing this like five or six years ago, when I tried to deploy to AWS and it was really complicated and there was just so much to do. I think the docs were out of date and it was just difficult. Then I tried Heroku, which is a platform as a service, and Heroku was much easier. I knew that there was the potential to scale up and I wasn't going to have bottlenecks and the only thing is I would perhaps pay a little more than if it was on AWS, but it was going to save me time. So I have not been shy about talking about how much I enjoy running my applications on Heroku, and I think there's a lot of people out there who like to do that as well, because it's a platform as a service that simplifies your life. There are other platform as a service. Heroku happened to be the one that I started using early on.

But when you think about the mapping from platform as a service to database as a service – So platform as a service solves some scalability problems. It solves some onboarding complexity. What are the parallel problems that a database as a service solves?

[00:50:44] AD: Let me just first take a step back and say, we actually coach a lot of our customers who are starting on-prem on thinking about what it means to shift to cloud. The key detail that's can so easily be missed is it's not just enough to move to cloud. Obviously, in your care and in my case, a lot of us just started cloud, but for folks who started on-prem, or for us who started in cloud, it's not just enough to be using cloud to be getting the benefits of what cloud really unlocks.

The key benefits that cloud really unlocks, the true return on investment comes from when you can leverage as a service offerings, frankly, wherever you can, because it allows you to go focus on your expertise, which is your own business and not focus on the kind of plumbing and all of the end-to-end expertise required to do those various layers of the stack.

It's easy to kind of write off the notion of, "I'm going to run my own database internally or run my own MongoDB or any other database. It's easy to kind of feel like, "Oh, it sounds straightforward. I'll just start some software and be done with it," but it's just not all that simple. It

becomes this giant technical debt. You have to think about infrastructure provisioning at the instance level, VPC level, the network connectivity configurations, security set up, TLS configuration, authentication, configuration, operating system patching, database level upgrades and patching. You have to think about just setting up your distributed systems. Thinking about ensuring that your applications are connecting to them all, and then you have to think about all of the upkeep of all of that and being able to be ready to do that in an automated manner every time you have anyone knew who wants to do anything more with your database, or spin up a new environment for your database.

All of that becomes just a huge time sink, and requires a lot of expertise that, frankly, a lot of us don't have. We don't necessarily all know how to ensure we have end-to-end security best practices built in. That's where something MongoDB Atlas which just has end-to-end authentication required, TLS required, firewall required, encryption arrest required. Can't shoot yourself in the foot as a result. All of those things are all about saving you time and allowing you to focus on your business.

As a service, really, what cloud really let you see the true benefits of cloud. Unfortunately, a lot of folks will treat cloud like their data center and manage a bunch of infrastructure in it like their data center and all of a sudden it really – It's not that different than what you were doing when you're using a co-location space. You're just trading it that way.

Now I will share that you're right, that MongoDB Atlas is sort of more like a platform as a service. In particular, it starts looking more that way with something they call MongoDB Stitch, which is actually layered on top of Atlas and is a serverless layer above MongoDB Atlas so you could run serverless functions there. You can also run triggers based on changes in the database, and we have something called stitch query anywhere which allows you to actually run MongoDB native queries directly from your clients from your JavaScript code. So you could sort of see MongoDB in a way becoming a bit more hybrid with our cloud platform, but we're still by no means a full-service platform as a service today.

[00:53:47] JM: This is actually something I wasn't planning on exploring, but it's an interesting area that I know people who are listening are curious about. So the serverless migration to serverless infrastructure or the movement towards serverless infrastructure, some of the is

standalone functions as a service that people stand up on Amazon Lambda, or Google cloud functions, and you just call these things ad hoc to just trigger some function. Other times there are other serverless platforms, like we've talked Auth0. Auth0 has a serverless platform that has triggers that you might want to tightly couple to your identity platform.

So you start to see – And we've talked to Cloudflare. Cloudflare has another serverless offering that they tightly couple to things that you would want at the edge of a CDN, for example. Here, you're saying there are potentially serverless functions that you want to be tightly coupled to your database logic, or perhaps geo-located close to your database. What are some examples of serverless functions that you might want to deploy close to your database? Whether we're talking about close logically, like this is logic that relates to my database and therefore I want to define some JavaScript function that's close to my database logically, or geographically, if you want to literally deploy it close to the database.

[00:55:26] AD: Yeah. I mean, I think there are lots of – Sky is really the limit on the permutations of this, because it's really all about saying, "I'm going to, with low latency, very close to the database, be able to execute compute," essentially, arbitrary functions that I could do.

So maybe I want to simply change or kick up a workflow based on a change in the database or a webhook comes in and I want to turn that into a new kind of object in my database, and then I want to kick off an SMS via Twilio service. All of these kinds, they're all related. Like you mentioned, Auth0. All of these as a service kind of serverless layers, they're all related and that you can start integrating them with each other to basically be able to do all kinds of interesting things that you need to do to build a modern application. Whether it's tapping into a quick voice to text service, or triggering an SMS or email, or doing a quick image recognition service, or anything under the entity, all of the above.

Building a modern application, you want to leverage all these things rather than build each of those things custom yourself. These serverless layers just allow you to more easily plug into those things without you having to run the boilerplate application code that does all of that interfacing. So it's more about – It's really all about, again, saving time, but it is a big mental shift.

[00:56:41] JM: Are you seeing people making that mental shift and going fully serverless or going in that direction?

[00:56:49] AD: The way we, for example, build Stitch above Atlas is all about this idea that you can have any mix of regular old app code running get your database and some set of services using Stitch, or you can go all-in on Stitch, and you could do everything in between. In other words, it doesn't hide the database which, frankly, some of the other serverless platforms are more about hiding the database, like you must use the platform.

Our approach is you can use MongoDB as much as you want directly, or you can go up one level and do functions in our serverless layer as well. What this means is a lot of folks are going to start small. They're going to have a pre-existing application and they need to add a new capability. Maybe it's, "I need to add in my image recognition software. I want to upload images and I want them to quickly get like an estimated image to text or something," right? You want to add that in. Maybe instead of having to put all the boilerplate into your standard app code, you start just leveraging a Stitch function for some of those capabilities and then you can over time add more capabilities that way. I think today we're still in the – Most people would do the mix mode rather than all-in for most applications.

[00:57:58] JM: Yeah. I think the usage of these kinds of things are going to really pick up eventually. I did an interview recently with this real estate AI company, really savvy engineers, and they started their company just in the last couple of years and it's entirely serverless and it's one of the first companies I've talked who they just don't even have the notion of – I mean, maybe in some edge cases, they're "running their own infrastructure", and even that sense, they're running it on a cloud provider. They're running it on AWS or running it on Google cloud, but they have gone all-in on managed services and functions as a service and in return they get such operational speed and high up time that they can just move faster with a smaller team. It's really amazing, because I did that interview fairly recently. I started hearing about these serverless things, I don't know, two, three years ago, and it just takes a while for it to really percolate and to make it – Yeah, I guess get wide adoption. Get acceptance. I guess people also still have to figure out like what are the patterns that we're going to be following here.

[00:59:13] AD: Exactly. Knowing how to appropriately set up test staging environments, knowing how to store your code in GitHub the right way for this new paradigm and how to get it in and out of the serverless environment where the functions are running from your repository. All these kinds of things are just new for people. Yeah, definitely exciting to watch, still early days.

Basically, if you're a technical person, you're very much a scarce resource in your company today. I think all of us, we find ourselves doing things that feel laborious on occasion and feel like they're kind of not strategic use of our time. It's easy to kind of not think about it at the individual level, but when you look at the big picture wise, your company really wins and loses on essentially making you productive as a developer.

Nowadays, developers are the kingmakers. Essentially every kind of company, because every kind of company, digital experiences are make or break for their brand. It's kind of easy to not feel or not notice, kind of like death by a thousand cuts, the cost of something you could be using that you're not yet using. But I think it is as developer community, there needs to be kind of a recognition of you actually have a lot of power and things that you demand that make you more productive, like moving data as a service type of offerings. Those capabilities, people are going to listen to what you want, because it's vital to the business.

[01:00:35] JM: We started by talking about performance and we've moved into talking about product development. I don't think we've really bridged those two. I guess to wrap up, could you talk about a particularly difficult product management or engineering challenge that you encountered while building Atlas?

[01:00:52] AD: It's difficult to say there's a specific item. I mean, there's an endless and – We've got, call it over 100 developers and product designers working full time on our cloud offering. So this is a big offerings. We're constantly iterating on it. Relentless, every three weeks, we have a new release with new capabilities. I think from a product perspective, the challenge is knowing how to optimize what should we build that will help the most of those three categories strategically the best, quickest, verse what are the things that are nice to haves that we're going to do later?

I'll give you a good example. I mean, one of the key things that we always knew when we built Atlas is a canonical MongoDB replica set has three nodes, and we're to support an Atlas. We knew from the very beginning we're going to support at least the big three public clouds, and we do today support the big three public clouds.

So we always kind of knew it will be just so awesome if we could just do MongoDB cluster that spanned all three clouds; Amazon, Google Cloud and Azure and it could withstand a full cloud outage. Yet this is the kind of thing where most of our customers are not really ready to take advantage of that, because their application infrastructure is not multi-cloud yet. They're all thinking they need to go there eventually.

As a result, we've focused Atlas on building great in-cloud cluster experiences we haven't gone multi-cloud yet for particular cluster. We have live migrations so you can move from one cloud to another with essentially no downtime. We have a live migrations so you can move from self-manager on-prem to Atlas with no downtime.

What we haven't done yet is true clusters that spanned Google, Azure and AWS. From a product perspective, it's a great example of it's a big project. We want to get there, but knowing exactly when we'll get there and when we'll be able to finish all the work required to get there. It's one of those things where if you go too early to do those giant product deliverables, things that most folks aren't yet ready, then you're doing so at the expense of a lot of your current customers who have a lot of other issues that you need to optimize. So that's a tough one that we've made a decision around sort of doing it a bit later and just focusing relentlessly today on all of our existing customers and their needs. But that's still quite a myriad set of needs you can imagine.

[01:03:02] JM: Very interesting. Yeah, and thinking about it from a strategic perspective of Mongo has a very successful business and then they decided we're going to create a cloud offering. I think it makes a lot of sense, because it's doubling down on the core competencies of the company and really focusing. Like the fact that there is 100 people working on this cloud offering, that's a serious dedication of resources to building a cloud service and it sounds like there are enough challenges to accommodate those hundred people.

[01:03:40] AD: Absolutely. From sort of a big picture perspective, not just looking at cloud. MongoDB, we think other than the fundamental data model being what developers from an impedance between the app and their data need today, all about agility and what they want today. Other than the data model [inaudible 01:03:55] just the core benefits of the distributed system which get you all the things we talked about earlier.

Really, the other key value proposition of MongoDB generally speaking is that you can run it anywhere. You could run it on your laptop. You can run it even on a mainframe if you can believe it. You could run it in your data center. Of course, we build to Atlas so that if you're running it in the public cloud, you could get a more exceptional than ever experience true as a service. But it's still the same database no matter where you run it. You're never locked in to particular cloud. You're never locked in to Atlas. You can always go make those changes, and we think that really about liberating our end-users so that they feel that MongoDB is really a liberating technology.

I think that that has some nice alignment to what's been happening around application portability with a lot of the trend towards Kubernetes and other platform as a service thinking at the app tier, which is basically allowing it to be easier than ever to move application infrastructure around stateless infrastructure. We think that those are a great combination. You can easily move your apps around and you can now easily move your database around with MongoDB having a consistent experience everywhere.

So we think running everywhere is very valuable to people, maybe not immediate value to a lot of folks, but a strategic long-term boardroom level, we think that's very valuable.

[01:05:12] JM: Okay, Andrew. It's been really are talking to you, and I look forward to seeing the developments in the future for MongoDB.

[01:05:19] AD: Jeff, thank you so much. It's really great. Thanks for having me on.

[END OF INTERVIEW]

[01:05:25] JM: Failure is unpredictable. You don't know when your system will break, but you know it will happen. Gremlin prepares for these outages. Gremlin provides resilience as a service using Chaos Engineering techniques pioneered at Netflix and Amazon. Prepare your team for disaster by proactively testing failure scenarios. Max out CPU, black hole or slowdown network traffic to a dependency. Terminate processes and hosts. Each of these shows how your system reacts allowing you to harden things before a production incident.

Check out Gremlin and get a free demo by going to gremlin.com/sedaily. That's gremlin.com/sedaily to get your free demo of how Gremlin can help you prepare with resilience as a service.

[END]