EPISODE 15

[INTRODUCTION]

[0:00:00.5] JM: A new programmer learns to build applications using data structures like a queue, a cache, or a database. Modern cloud applications are built using more sophisticated tools, like Redis, or Kafka, or Amazon S3, and these tools do multiple things well. They often have overlapping functionality with each other and application architecture becomes less straightforward should you use Kafka for your queue, should you use Redis for your queue, I don't know.

These applications that we're building today are data intensive, rather than compute intensive. Netflix needs to know how to store and cache large video files and stream them to users quickly, and Twitter needs to update user newsfeed with a fan-out of the president's latest tweet. These operations, they're simple with small amounts of data, but they become complicated once you have a high volume of users and a high volume of data, and that's the idea of the data intensive application.

Martin Kleppmann is the author of a book from O'Reilly called *Data-Intensive Applications,* and this book is about how to use modern data tools to solve modern data problems. His book includes high level discussions about architectural strategy and lower level discussions like how leader election algorithms can create problems for a data intensive application.

If you're interested in hosting a show for Software Engineering Daily, we are looking for engineers and journalists and hackers who want to work with us on content. It's a paid opportunity. We do around $300 paid out per episode and you can go to softwareengineeringdaily.com/host to find out more.

Also, the Software Engineering Daily Store is now open. If you want to buy some Software Engineering Daily branded t-shirts or hoodies or mugs and support the show.

Now, let's get on with the show.

[SPONSOR MESSAGE]

[0:02:06.5] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance, it has never quite gotten the due that it deserves. Today's dynamic applications, hybrid clouds and volatile internet, demand that you rethink the strategic value and importance of your DNS choices.

Oracle Dyn provides DNS that is as dynamic and intelligent as your applications. Dyn DNS gets your users to the right cloud service, the right CDN, or the right datacenter using intelligent response to steer traffic based on business policies as well as real time internet conditions, like the security and the performance of the network path.

Dyn maps all internet pathways every 24 seconds via more than 500 million traceroutes. This is the equivalent of seven light years of distance, or 1.7 billion times around the circumference of the earth. With over 10 years of experience supporting the likes of Netflix, Twitter, Zappos, Etsy, and Salesforce, Dyn can scale to meet the demand of the largest web applications.

Get started with a free 30-day trial for your application by going to dyn.com/sedaily. After the free trial, Dyn's developer plans start at just $7 a month for world-class DNS. Rethink DNS, go to dyn.com/sedaily to learn more and get your free trial of Dyn DNS.

[INTERVIEW]

[0:04:01.9] JM: Martin Kleppmann is the author of *Data-Intensive Applications,* a book from O'Reilly. Martin, welcome to Software Engineering Daily.

[0:04:09.2] MK: Thanks so much Jeffrey for having me.

[0:04:11.0] JM: Many applications today are data-intensive rather than compute-intensive. Explain the distinction between these two application types.

[0:04:22.9] MK: Sure. I would call an application data-intensive if data is the primary challenge that arises when building that application. That could be, for example, the amount of data, the

sheer volume of it, but it might also be the complexity of the data, how interlinked it is, or might be how fast it's changing both in terms of the data writes, but also the structural changes.

Quite a lot of applications which fall in that category — My background is from doing data infrastructure to internet companies. I used to work at LinkedIn and the kind of architectures, they are somewhat similar at Facebook, Twitter, Google, Amazon, Microsoft, et cetera. I believe there's actually similar kind of data processing and data storage challenges occur in many other areas as well, which I'm less familiar with.

One area that came to mind, for example, would be scientific data. If you're a physicist at The Large Hadron Collider, for example, I believe you might be doing a lot of data crunching in order to discover new particles. If you're a genomics researcher trying to find a new cancer drug then maybe you would be crunching through large amounts of genome data. So I think it's quite a generic term data-intensive applications just because there's so many different things that really fall under that category.

[0:05:52.2] JM: We had this web 2.0 period where there were companies like LinkedIn and Twitter and these companies were getting started and they had to deal with these high volumes of data. During this period of time, we saw the creation of a lot of abstractions that make it a whole lot easier to work with some of the canonical problems of data-intensive applications.

We saw things like Kafka get developed. Obviously, AWS got its start in this period. Now, we're in this world where we've got some really useful abstractions that allow us to solve some of the canonical problems. Then, if you're Netflix, or Uber, or a company of that scale where it's like you not only have the problems and the challenges of a Twitter or a LinkedIn, but you maybe got these new challenges where you've got — It's obviously a kind of a buzz word. It doesn't really make any sense, but the real time data challenges.

They're slightly different challenges, but they are a superset of the challenges that came before. We are in this kind of new era where not only do we have the big data stuff of web 2.0, but we have the fast data challenges of maybe web 3.0.

[0:07:13.1] MK: Yes. I don't know exactly which version number of the web we are at right now, but I think this general idea of new challenges having risen like maybe in the last 10, 15 years or so I think is very true. I actually spent a fair amount of time in preparing this book, just going back a bit through computing history and looking at the history of, say, relational databases which were developed in the 70s and really stayed around for a long time and were really the dominant way of data management for decades, literally.

Then, suddenly, people thought of moving away from it. Then, at the moment, no SQL came up in 2005 or something like that. Really, that was not so much a movement against relational databases. It's rather embracing a much wider range of other kind of storage and processing technologies that's previously been not even considered. Part of that was definitely driven by just relational databases no longer really fitting all of the different use cases, all of the different things people wanted to do with data. Although there are still super relevant relational databases. They're still used a lot. They're not the last work in terms of dealing with data.

Since you mentioned abstractions, I think relational databases were so successful because they were a great abstraction for the kinds of data that they were designed for. The problem is that now we have things, like you mentioned, is real-time data, for example, where at least the classic implementations of relational databases simply don't handle that well.

I feel like at the moment we're searching for these new abstractions that we'll see us through the next couple of decades. Right now, everything is — That there's this explosion of many different technologies and they're all competing for getting our attention, and it's really hard to get an overview sometimes or what technology is actually suitable for what purpose.

[0:09:20.6] JM: It is, because you write about this. The idea that when a new programmer starts out and you're taking your basic data structures class and these data structures are like a queue, or a database, or a cache, and these are data structures with very clear responsibilities, but there are these newer tools, like Redis, or Kafka, although these tools are now like a decade old. They're less defined in their core functionality, because they can operate as both a message queue and a data store, or they might have these varying durability guarantees where it's not as easy to explain the durability, or is easy to understand the durability, the consistency policies of them.

We have these newer data systems and they have some overlapping functionalities. It seems to me like these fundamentally make things just easier, because, usually you have something like Redis and you have it doing some small subset of what Redis could potentially do, and then maybe in the future you get to leverage additional aspects of Redis, but it's not like — I don't know. What are your thoughts on how to approach these abstractions that provide a large and often overlapping with each other set of functionality?

[0:10:43.8] MK: I think it's very much an open question actually. What I tried to do in this book is to outline the different approaches that exist kind of on a fundamental level and which pieces of software implement which approach and then try to analyze how we can compose them together to building more complex applications.

You're right. If you take something like Redis, it offers an API with a certain interface. Take something like Kafka, again, it offers a certain API. They're kind of similar in some ways and very different in other ways. Most of us simply don't have a good understanding of exactly what we're getting from these tools beyond the surface level of the API.

I think with the in-memory data structures like you mentioned, like a linked list or an array or something like that, we understand how to deal with them, because we've been programming with them for decades. You know, for example, if you have an array, then accessing a random element is cheap. It's like auto 1, but removing the first element is auto N because you have to move all of the remaining elements. Whereas with the linked list, it's the other way around, like removing the first element is really cheap, but accessing an element in the middle is expensive.

We've got these like engrained ways of thinking about these abstract data structures that help us figure out which data structure you might use and which circumstances. Now, when it comes to these new data tools, I'm not sure we yet quite have the language to reason about exactly what we're doing, what we're getting. At the moment, you get things like, "Okay. Something writes to disc. Okay. When does it write to disc? Does it guarantee that it has hit the disc at the point when it acknowledges something? What happens if that disc dies? Does it guarantee that it's replicated to another machine? Does that replication happens synchronously or asynchronously?" et cetera, et cetera.

There's a whole lot of complex challenges there and part of what I'm trying to do in this book is to just put them into a systematic language so that we can even talk about them and compare them.

[0:12:55.9] JM: The Big O notation for writing to a data store that's distributed is not as easy to determine when you have nodes dying. Even if nodes don't die, even if you're just doing partitioning, and sharding, and replication and O, but actually we want this to be an ACID transaction. What's the Big O of making it an ACID transaction? These things are not well-defined. Let's talk —

[0:13:35.0] MK: Yeah, I could imagine that maybe we'll get better ways of reasoning about those findings and how the right way of summarizing the essence of how expensive something is. I made one attempt in a research paper a while ago that was criticizing to CAP Theorem. People site that a lot in the context of building distributed systems. I personally find the CAP Theorem not at all useful.

I was arguing instead we should reason about whether something requires a network round trip or not, essentially. How dependent is something on network delay, which seems to me like a kind of nice way of boiling something down to the essence, because there are some things that fundamentally you can do without talking over the network and other things you fundamentally cannot. That makes a nice dividing line between different consistency models, for example.

Yeah, these sort of ideas are nowhere near as established as the Big O notation. I think it's going to still take us a while to really figure out how to best formulate those things.

[SPONSOR MESSAGE]

[0:14:48.5] JM: Software engineers know that saving time means saving money. Save time on your accounting solution. Use FreshBooks Cloud Accounting Software. FreshBooks makes easy accounting software with a friendly UI that transforms how entrepreneurs and small business owners deal with a day-to-day paperwork. Get ready for the simplest way to be more productive and organized. Most importantly, get paid quickly.

FreshBooks is not only easy to use, it's also packed full of powerful features. From the visually appealing dashboard, you can see outstanding revenue, spending, reports, a notification center, and action items at a glance. Create and send invoices in less than 30 seconds. Set up online payments with just a couple of clicks. Your clients can pay by credit card straight from their invoice. If you send an invoice, you can see when the client has received and opened that invoice.

FreshBooks is also known for award-winning customer service and a real live person usually answers the phone in three rings or less. FreshBooks is offering a 30-day unrestricted free trial to Software Engineering Daily listeners. To claim it, just go to freshbooks.com/sed and enter Software Engineering Daily in the How Did You Hear About Us section. Again, that's freshbooks.com/sed.

Thanks to FreshBooks for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:16:25.7] JM: When you talk about the difference between looking at a distributed system through the lens of CAP Theorem versus the question of whether something is going to make a network round trip or not, I guess — I kind of like your approach, because CAP — I guess your approach of thinking in terms of whether the network is going to make a round trip or not, that sort of simplifies things.

I guess with CAP, CAP doesn't really — Talk more about that. What's with the criticism of the CAP Theorem? I always found it to be a useful simplifying way of explaining, "Here are the tradeoffs between a distributed system and you can never give up partition tolerance." I don't know. Talk more about the distinction between your two paradigms.

[0:17:27.7] MK: Yeah, certainly. The problem with CAP I see it is mainly that the theorem, as it's formally defined, doesn't actually say what most people intuitively think and say. It's simply a different statement. The theorem that talks about a very specific model of consistency, which is linearizability, which I explained in the book in some detail. It talks about a very specific notion of

availability namely availability in the context of arbitrary network interruptions that last forever. For example, it doesn't speak about availability in the context of a network that is interrupted and then repaired. Again, it only talks about it in the context of partition that lasts forever.

The definition of availability in there is also kind of not what people expect, because it doesn't take time into account. It considers a service to be available if it responds 10 minutes later, which is not actually what most people would consider available, but that's the definition that the theorem uses.

I feel it kind of misses the reality of what people actually talk about, but then if you redefine the terms and say, "Actually, we mean something different with availability," well then the theorem no longer applies in unchartered territory again.

If you want to use it as a theorem, as a mathematical truth, then you have to be precise, but people are not precise when they talk about the CAP Theorem.

[0:19:01.5] JM: Let's talk more about your book. You encourage readers to be able to describe the load on their system. In order to talk about scalability, you need to talk about what is the load on your system that your scalable response is based off of. What are some different ways that load can manifest on an application?

[0:19:27.4] MK: It depends very much on the application in question. In some cases, the load might be just the volume of data you're dealing with, or the number of requests per second that are coming in, or maybe the number of writes per second, maybe some types of requests are more expensive than others, so you care more about those.

It depends very much on what you're trying to do, but then once you have some load number or something that you can put a number on, then you can start reasoning about, "What happens if that number doubles? How do we respond to that?" Because then at some point you'll probably get to the point where a single server can no longer handle the load. In that case, you've got to, somehow, distribute it across multiple servers or you've got to accept that a lot of things are doing to slow down.

If you want to keep the performance constant and you're probably going to have to add resources somehow. Reasoning about scalability is just reasoning about the process how you can address an increased load by adding more resources. Of course, that depends on the type of load that it is, that you're dealing with. Also, the structure of the data, what kind of things you're trying to do with it.

Really, it's not a one dimensional thing. It's meaningless to say that something is scalable or something is not scalable. Instead, really, we should be talking about, "This system can handle an increase in this kind of load by doing this and that." Then, you can start having an educated conversation.

[0:21:01.0] JM: We will work towards a discussion of the distributed system architecture for a data-intensive application. Let's start with the idea of the data model. The data model is the representation of how we store and access the data of our application. Some applications use a document model. Some use a relational model. Many applications use some combination of the two. How should an application architect reason about the data model of an application?

[0:21:37.5] MK: I would add to those two models a third one, which is graph data model, because I think if you look at the three one side by side; that is document and relational and graph, you see some interesting characteristic differences. I'd say the biggest difference has to do with the structure of the data. That is you've got some kind of things that have a relationship to each other, and from relational modeling we know that you can have one to many, or many to one, or one to one, or many to many relationships.

The type of relationships that are dominant in your data really determine the type of data model that's best suited for it. A document model, I would say, fits very well if you've mostly got one to many relationships. The example I use is, for example, a LinkedIn profile which is — It's like a resume essentially. You've got a profile that represents one person, and on that one profile, you have a name and you have the headline, and then you have multiple jobs that you've worked on, and multiple periods of education, and multiple social media accounts maybe.

Then, you've got this one to many relationships between the jobs and a profile, and that is each job belongs to exactly one profile, but each profile may have many jobs. This kind of thing you

can represent very naturally as a JSON document, say, by just having a document profile and within that is a list of jobs. That kind of thing you can of course represent in a relational model as well, but it gets clumsier especially once you have things nested within other things and you need to represent order in some way, so you then have some kind of ordering column attached to things and then you need some really complicated queries to try and pull all of these things together or you end up doing multiple queries in application codes.

If that's the structure of the data, then just fetching a single JSON blob is great. Also, if you have a lot of data with different fields depending on where the data is coming from, then splitting it across 20 different tables in a relational data model may be crazy. In that case, again, putting it in just a typeless JSON document might be by far the simplest.

Then, on the other hand, if you have a lot of cross-referencing between different objects, or call them documents, you might have a reference from one document to another, then actually that starts looking quite similar to a relational model. Very quickly, once you end up with these many to one, or many to many relationships, you end up with something that's essentially a relational database except without all of the nice query languages.

Then, if you go even further towards the many to many model, you end up with graphs. I see that really as a generalization of a relational model where, really, in principle, anything can be related to anything else, and so you can keep adding to the graph by introducing new types of vertices and edges.

[0:24:50.8] JM: In the early days of an application, whether it's like a social media app or a SAS with a front-end, you often have a front-end web app, or a mobile app that needs to interface with some sort of backend data store. These data stores are growing in volume. The query complexity that the front-end might make, it's still pretty simple, but you could imagine it growing in complexity. There are these different query types that the front-end apps are making, it's changing — You have these declarative syntax that is growing in popularity in front-end web applications. Declarative syntax is also growing in popularity for things like Kubernetes.

Then, you also have these middleware layers for querying, like GraphQL, and you're just really seeing a growing richness between what we used to call front-end and backend. You're getting

all these middleware stuff that can be really nice to work with and can add some flexibility. At the same time, these data stores are growing in volumes, so it kind of makes sense to build up these different middleware layers that make it easier to interface with the data lake.

How has the strategy for the — Basically, the query between the front-end and the backend as we get these thickening layers of middleware between these two ends. How does the strategy of a data-intensive application architect change?

[0:26:42.3] MK: I think it seems like a really good idea to me to introduce these abstraction layers. Something like GraphQL essentially gives you a fairly unified abstraction for a client device to talk to some server-side system. At least for the types of use cases that it targets, it seems to do that very well. It just kind of gives a common pattern for people that has been proven in practice and it allows you to then change the implementations on either side of that interface either on the client side and on the server-side, so you can swap out, for example, one database to another under the hood and still keep the same GraphQL interface layer.

That kind of thing is great for evolving an application, and so I think it's very good to be conscious of the interfaces and how long you're going to have to support them. If it's something like the interface between a mobile app and a server, then if somebody never upgrades the mobile app on the device, then you're going to have potentially very old clients accessing a server-side, although you can upgrade to server-side easily, that's not the case for the clients. You've got a long compatibility history to worry about there.

If you can change the storage on the server-side independently, then that's a great asset, because it means you can just start off with a simple relational database. For example, initially, keep evolving that in the early stages of your application, and then the more you understand about exactly what kind of access patterns you have, what kind of data you have, you might find that maybe some more specialized tools are more appropriate and you can then still always migrate to those and you can do that without having to change the communication between the client, that is the mobile apps and the server-side. I think that's a great way of building systems to be maintainable in the long run.

[0:28:42.3] JM: We've done shows about Apache Avro and Parquet, and in these shows we discussed the tradeoffs between columnar and row-based data storage format. Whether a dataset is in memory, or it's on disc, you could put it in a row-wise format, or a columnar format. What are some guidelines for when people should store in row-wise format, or in columnar format?

[0:29:12.8] MK: I say as a quick rule of thumb, if you need to scan over large numbers of records but you're only interested in a small number of fields from each record, then columnar is better. Typically, that kind of workload arises in analytics applications where you want to do some kind of big aggregation of some more average or maximum of some column over some large number of rows group by X aggregating on the values of X.

With that kind of thing, if you're going to be scanning across a hundred million rows, then doing that in a row-oriented format can get quite slow because you just have to pull a huge amount of data of disc, and columnar storage is much faster there because it can represent individual columns much more compactly and it only has to read the columns that you're interested in.

Where the column storage falls down is, if course if you need to update, because now every single write has to go to lots of different places. That way, in most column stores, you actually get a kind of two-step process where if you write to it, they'll first go to a row store and then once it's accumulated enough data in the row store then it will flatten those out into columns and merge them with the rest of the columns.

Typically, I see the row-oriented data being used in analytics where most of the access is read only and the access it typically over large numbers of rows in one go. On the other hand, row oriented tends to be used in places where you have lots of random access reads of individual rows and lots of random writes, and so those are much more the kind of OLTP transaction processing type workloads.

[0:31:09.5] JM: There is the data encoding format as well. There's the data storage format, which is the row-wise or the columnar format. Explain what the encoding format is and what the relationship between the encoding format and the storage format could be.

[0:31:27.4] MK: You mean the encoding that is used physically by the storage engine, or do you mean from an application point of view?

[0:31:33.0] JM: Okay. Maybe I'm confused about this, because I haven't done much about this — I've done enough shows about this to know much what I'm talking about, but I know there's Parquet and Thrift and Avro, and so — I don't know. I guess I'm confused about this stuff. Maybe I should have done some more research about these different things beforehand. What's the relationship between these things, like Parquet and Avro and Thrift and the columnar versus row-wise question?

[0:32:01.9] MK: Right. All of those you mentioned are format that's store their data in files to a first approximation. Things like Thrift also define an RPC protocol for going over the network, but when you're talking about data storage, their way of taking some records in some formats typically records that conform to some scheme and writing them out as a file. That would be used often, for example, in the Hadoop context where you have HTF, the Hadoop file system which stores files, and so you have to put your data in files in order to store that. You have to encode it as a sequence of bytes somehow. Which is different from the database abstraction typically, where if you use postgres, for example, what you get is a query interface, you don't get a file format.

There are files on discs somewhere internally behind it, but actually the format they're in is some kind of obscure binary thing that only postgres understands. You can't really go and move the postgres data files yourself except by going through the SQL query interface. The same is true for many data warehouses, for example, which use columnar storage internally.

In that terms, you can have the row-oriented data layout that's provided, for example, by Thrift and Avro data files or the columnar data format from Parquet if you want to encode in files yourself. Then, a lot of storage engines from postgres under row-oriented side, or like Vertica on the column-oriented side, they will actually use their own internal storage formats. Similarly, you can group those into row or column-oriented.

[0:33:47.7] JM: If you're architecting the entire application, are you often making multiple versions of the data available? You have one copy of the data that is in a columnar format so

that it's accessible to large scan jobs, like finding the sum of a bunch of transactions or the sum of all of the — Or the average age of all of the users. You can answer queries like that, but then you also have the same data available in a row-wise format so that you can get all the data about a specific account so you can load a user's profile. Do you have that data basically copied in two different places?

[0:34:34.1] MK: Yes. That's typically what happens. The traditional way by which that happens is called ETL, that's the extract transform load process where you have some data that's being maintained and operational databases which will be typically row-oriented stores. Any kind of random access reads from the website or any writes from the users, they will go there. Then, you have this ETL process which extracts the data from those row-oriented databases.

Maybe transforms it into a different scheme that's more amenable to analytics, and then loads it into a data warehouse, and these data warehouses will typically use a column-oriented storage because it's optimized for a different access pattern. It's optimized for a business analyst who will sit there perhaps with like a graphical tool where they can slice and dice the data and analyze transaction history and so on. They'll be doing much more this kind of large scan.

We see here this pattern of the same data represented in two different formats for different access patterns, and that I think is a more general pattern that we actually see across many different aspects of data systems. Another example would be, for example, if you need to do full text search on some data, typically what you would do is you have the primary copy of your data which might be, say, products in a product catalog, it will have to primary copy in a database which will be an OLTP row-oriented database. Then, you have a copy of that data in your full text search index which might be like elastic search, or solo or something like that.

Again, you some kind of data synchronization of copying process by where every time you change the description of a product in the product catalog, in the database, you then also reflect that change into search index so that people can find the new description by keyword.

If you even think about, actually, this kind of representing data in multiple different ways, that even appears internally in a database. In any relational database, if you have a secondary index, what you're really saying is, "I want to construct this additional structure on the side,"

which is typically a B3 or something like that, “and every time you write to some table to update that index, to also allow me to find records with a certain value.” That’s what a secondary index is. It just happens to be done internally by the database.

Really, it’s just another way of saying we’re going to take the same data and have multiple copies of it represented in different ways, either sorted in a different way, or with a different storage layout. That allows us to access the data in ways depending on what you’re trying to do. In the case of a secondary index, depending on which column you’re searching by, in the case of a full text search index, that’s just search by keyword. In the case of a data warehouse and a column store, in order to do a large scan of all of those records.

I just see that as a general pattern of quite often we have a primary copy of the data in one system and that might be called the system of records, or the source of truth. Then, we have a whole bunch of different derived data systems which they take their data as a copy from this primary system, the system of record. Transform it in some way and then represent it differently in order to satisfy certain read access patterns. The pattern we have there is the writes or go to the primary storage system and all of the other systems are derived from it. The other systems only serve read requests.

[0:38:29.7] JM: This even happens on a single machine. Last night, I saved a file and then I opened up Searchlight, or Spotlight on Apple and I searched for that file because I had the file in one application and I wanted to open it in a different one. I searched for it and I was like, “Why can’t I find it? Why isn’t it visible?” it was because the search index hadn’t been updated. This kind of lag happens even on a single machine.

[0:38:57.0] MK: Eventual consistency.

[0:38:58.3] JM: It is eventual consistency.

[0:39:01.7] MK: You can’t even blame the CAP Theorem.

[0:39:02.6] JM: That’s just what I was thinking. You can’t blame the CAP Theorem.

[0:39:08.0] MK: You can't blame the network interrupting between the Spotlight search index and your hard disc file system. It's really a performance issue.

[0:39:16.8] JM: This is something I wanted to ask you about, because you wrote in one part of the book where you're talking about; we don't have to think about as much about the hardware messing up on a single machine. We don't think about, "Oh, this transistor is messed up and so we have to prepare for this hardware error on a single box." We do have to worry about flaky networks. We have to worry about flaky clock representations, like a representation of time might be messed up. We have to worry about nodes failing.

Presumably, there was a time where a "single box" had these sorts of issues; a vacuum tube burns out, and like, "Oops! Okay, the single machine doesn't work." Do you think we're going to get to a point where these types of the network flakiness or the clock flakiness where these things will be ideas of the past and it will be a lot easier to reason about these distributed systems, or do you think working out the network problems is just something we're never going to have figured out?

[0:40:22.8] MK: Frankly, I think it's fundamental and we're not going to be able to solve it. In certain limited domains, you can take the approach of treating everything as a single system. In the high performance computing world, these are the people who build super computers which are used for weather simulations, for example, they do take lots of individual computers, like lots of CPUs, lots of RAMs, join them up with a fast network and treat the entire thing basically as a single computer. What makes it basically a single computer is that if any one of those components fails, they just stop the entire cluster, repair that component and then continue the job.

This is exactly what happens in a single computer, like if your laptop — If a transistor decides to go nuts in your CPU, what you'll probably get is a kernel panic or some kind of crash and you'll just reboot the computer and restart again from the last saved state. You get exactly the same pattern there, but if any fault occurs in any part of the system, then you simply escalated through a whole system fault and just restart the entire thing. That works up to a point, but it depends of course how frequent the faults in individual parts of the system are. The bigger your system gets, the more likely it is that any one of those parts is faulty.

Now, there's a countertrend that, overtime, probably the parts get more reliable. As you said, transistors are more reliable than vacuum tubes of the past. It's quite likely that that trends towards individual components getting more reliable will continue. That said, we are approaching lots of strange quantum effects with the size of structures that people are getting to with silicon. Maybe that's not even necessarily true.

Assume that we have the kind of single node redundancy sorted out. You then still have to problem that at some point a failure will occur and then if your response to something going wrong is to just stop the entire cluster and reboot the entire thing, well, then as soon as any single part of it goes wrong, you would then have to interrupt service for the entire thing. That was possible in the days where, say, a business works only 9 to 5 within a single time zone and then can perform maintenance overnight. Now, with things that are connected to the internet, people expect it to be online 24/7. You actually never have that time when really you can just take the system down, stop the entire thing, reboot it, bring it back up again.

Instead, people actually want systems to be continuously available. That then implies that you have to be able to tolerate the faults of individual nodes because you have to have the frequency of a failure of the entire system be lower than the frequency of the failure of a single node. Whereas if you build your system in way that resembles a single computer, then the frequency of failure of the entire system is at least as great as the failure rate of a single node.

What we can get from being able to tolerate failures is the ability for the entire system to have better uptime than any single part of it. That's really, I think, the fundamental thing about distributed systems is it's some way something that we inflict upon our self because we want to build more reliable systems. If we didn't care about reliability, we could just treat everything as a single computer like, in fact, high performance computing people do, because with the weather simulation, for example, it's actually okay.

It's not an online service. It's a batch job. You start it and it runs for a couple of hours and you can write up checkpoints from time to time. Then, if something fails, oh well, you just reboot it and start from the last checkpoint. That's not the case with an eCommerce online shop, for

example, where you have to be processing transactions all the time. It doesn't make sense there to go back to a checkpoint from 10 minutes ago.

[0:44:45.2] JM: In dealing with these problems of distributed systems, we're obviously doing replication. Whether we are replicating to a search index so that we can look up a piece of data more quickly, or we are replicating so that if a node fails, we still have access to that data. If you're saying to a database, you want some fault tolerance incase that database node just burns out and you want to still have access to that other database node.

The replication issue, that's a canonical thing and replication is going to require us to — Across a distributed system, is going to require us to think about the Paxos-like roles of different compute nodes. Nodes can die. You've got a leader election that might need to take place, and I need all of these to occur while my application keeps up-to-date and the replicated nodes are up-to-date or up-to-date enough.

Explain how the challenges of leader election the Paxos-like issues, what challenges can those create for a data-intensive application?

[0:46:00.3] MK: The attempt of something like Paxos, so consensus algorithms is really to create a bit more reliable abstraction than what the distributed systems gives you by itself, because it's really a nightmare to try and deal with the system where you have absolutely no guarantees about which node is actually alive, which is dead, are they just temporarily not responding? Are they going to come back again later? Did my data get there? You sent a packet there, you didn't get a response. Did it arrive? You don't know. Maybe it just got delayed somewhere.

If you're programming distributed systems at this basic level, it's really difficult just because there are so many things you can consider, you need to consider if you want to get it right. The idea of consensus algorithms like Paxos is to give you a bit stronger guarantee. The guarantee it gives you with consensus is that you get some nodes to decide on something and once they've decided, they won't change their mind. That sounds really basic, but even just achieving that turns out to be quite tricky in practice.

At least we do have implementations of consensus algorithms, and so it's used in things like Apache ZooKepper, for example, which is then used to often build cluster management things. That is for a database cluster, for example. You might want to ensure that at any time, there's only a single primary, a single leader for every partition. That's important, because if you have two leaders for the same partition, then that means they're going to both accept writes and they're going to diverge, and so you've got what is called split-brain.

Having a consensus system allows you to avoid this kind of split-brain thing just by delegating the consensus problem down to an algorithm that other people have thought about really hard and the higher level applications can then use stronger guarantees. For example, that there will only be one leader at any one time.

Still, there's getting these kind of things right so that they really always satisfy the guarantees that you think they do is still tricky. I think we still have an education challenge there of just knowing exactly what you can rely on and what you cannot rely on in a distributed system.

[0:48:28.0] JM: If I have a large volume of data, I need to partition it, and partitioning is going to keep the access fast and it breaks a large volume of data up so that it can be spread across many nodes. Then, you have this whole process of indexing the different partitions and you have to look up the right partition and then the right document within the partition and all of these partitioning stuff is occurring in addition to the replication. You break a large volume of data into a bunch of partitions and then you replicate all the partitions.

Describe the challenges of architecting a scalable partitioning strategy, because at a certain scale, even just the operation of looking up the right partition can be a time consuming process.

[0:49:22.8] MK: It is tricky especially with data that highly interconnected. If your data is just key value structure and you always know what the key is and you just want to read or write the value associated with the key, the partitioning is basically trivial, because typically what people will do is use some kind of has function to map the key to a particular partition. Even the client can just compute that hash function and then look up in a table, "Okay. Partition 15; I need to contact this IP address on this port number," and then contact the right node.

It gets a lot trickier even if you just want to say a secondary index, because then if you're looking for a certain value, you're looking for the value that appears within a record somewhere, like you're looking for all of the cars in your database that are green in color. Then, in one case, either you have to index on the same nodes as the record that's being indexed. In that case, writing is easy because when you update a document, you only need to update the index that's on the same node. Now, whenever you read, you have to go to all partitions, because a green car that might be in any of the partitions, you have to query all of them in a scatter-gather kind of process in order to find all of the possible occurrences of green cars in your database.

Now, on the other hand you could structure the secondary index the other way around that actually you make sure that all of the index entries for green cars go on to the same node, but now the writes get more complicated, because now you have whenever a document gets updated, or a document gets added with a particular color, then the secondary index for that document may actually be on the different node, so you now have to update that as well and now you have to worry about what if one request succeeds and the other fails, then your two nodes get inconsistent with each other.

Partitioning gets hard in those kind of cases where you want to look up by something that is not just a primary key. You get a variant of this problem with graph structure data, and in graph, if you want to be able to follow an edge in the graph in both directions, you've got to be able to look up an edge in two different ways, once from the head and once from the tail, which is essentially the same problem as what you get with a secondary index. Those aspects of the data structure, when you have these interconnections, references from one thing to another or secondary indexes, the make the partitioning hard.

[SPONSOR MESSAGE]

[0:52:15.5] JM: Don't let your database be a black box. Drill down into the metrics of your database with one second granularity. VividCortex provides database monitoring for MySQL, Postgres, Redis, MongoDB, and Amazon Aurora. Database uptime, efficiency, and performance can all be measured using VividCortex.

VividCortex uses patented algorithms to analyze and surface relevant insights so users can be proactive and fix performance problems before customers are impacted.

If you have a database that you would like to monitor more closely, check out vividcortex.com/sedaily. GitHub, DigitalOcean, and Yelp all use VividCortex to understand database performance. Learn more at vividcortex.com/sedaily and request a demo.

Thank you to VividCortex for being a repeat sponsor of Software Engineering Daily. It means so much to us.

[INTERVIEW CONTINUED]

[00:53:28.1] JM: Where we're going with this is that whether you're talking about a read or a write, the idea of a transaction is, as you put it, slippery in your book, because if you're reading, does that mean that you're reading from the columnar store? Does it mean that you're reading from a cache? Does it mean that you're reading from a search index, and are you trying to read the most up-to-date version of an entry? Then, if you're writing, is the concept of a transaction – Is it the write to the up-to-date of source of truth data store, or is it write to the up-to-date source of truth data store in addition to the search index and the cache and the other LRU, and so it's like you have to define – If we're talking about a transaction, you have to define what do you mean by that transaction. How many different databases are you talking about in the sum of your transaction?

Then, some transactions are prone to raise conditions that might need to be serialized, because if you want to be able to look up how much money is in your bank account and if you're a stock trader and you're making all these trades, and trades are being scheduled and then they're going off at different times when the market hits a certain amount. You want to know an up-to-date amount of your bank account balance. In that situation, you probably want the transactions to that bank account balance materialize view to be serialized.

How do we achieve serialization in a big distributed system where you have these things like replication, and partitioning, and slippery transactions. How do you get the serialization down?

[00:55:19.9] MK: It's a really tricky problem. We do have distributed transactions that can operate across multiple different storage systems, like XA transactions have been around for a long time and a reasonable number of systems support them. Many relational databases do – If you're using the Java transaction API, for example, that's basically the same as XA. You can do that connected with message queues, for example, so you can do things like – Acknowledge a message in a message queue only if the transaction that wrote the processing of that message to a database was also successful.

You can get this atomicity across multiple different storage systems. It does come at a very high cost. It is possible to make these distributed transactions work, but the performance characteristics are really poor. Also, the full tolerance characteristics are poor, because you just need one part of the system to be doing slow or have a temporary outage and the entire thing grinds to a halt.

An approach that I found more useful actually in thinking about these serialization things is instead to think about the flow of data as it flows through a system in multiple stages. I mentioned earlier this idea of a primary or a source of truth data store and then other derived data stores. You could, for example, make a guarantee that you will only ever write to your source of truth and everything else is derived from that.

Now, this means that maybe that the revision happens asynchronously. Maybe it happens synchronously. You can decide that separately. You know that you could always rebuild your derived data store from the original – From the source of truth. It's just a purely deterministic transformation process and that means now that rather than having to do a write across multiple different data stores and have to ensure that that happens atomically, you can actually just do an atomic write to a single data store, which is your source of truth and then have everything else be derived from that.

Now, that gives you most of the full tolerance properties that you need, namely that you essentially get atomicity that is if you write to the primary to your source of truth, then that will also be reflected on the others, but you don't get the synchronicity necessarily. That is if you make a write to the source of truth and then immediately read from some derived data store, then that might not yet be up-to-date.

What I would assert is that, actually, in many circumstances, that is okay, and so you can think about it from a business point of view of exactly for which things you need data that is up-to-date to which degree. You can then make a tradeoff. You can say, “Okay. For some things, we really wanted to be up-to-date and we’re going to tolerate the lower full tolerance and the lower performance that comes with having up-to-date.

In that case, you can do a round trip through your derived data system and essentially treat the write as completely only once it’s been reflected in the derived system as well, which you could do by plumbing several stream processing stages together.

On the other hand, you might actually say, “Maybe it’s okay to just read a value that might be outdated by up to two seconds and you’re going to monitor how far behind it is so that it doesn’t go over two seconds. That will mean that, occasionally, in some edge cases, we violate some constraints. It could be that, “Oh! Accidentally, we sell more items than we have in the warehouse,” or, accidentally, you sell the same seat on an airplane or the same seat in a theater to two different people. That’s not going to happen very often. Occasionally, it might happen.

In that case, you’re just going to apologize to the customer and you’re going to send them a voucher and say, “Oh, sorry. Would you mind choosing a different seat, or would you mind waiting for an extra two weeks until we’ve got that item in stock again?”

That’s exactly what would have to happen in other circumstances anyway. If you have a warehouse and you know that you have a certain amount of inventory in the warehouse, what happens if worker in the warehouse drops an item and breaks it, or a forklift truck runs over it? Then you’ve got one few item than you originally thought you had. If you have to orders for the last two items and one of those two items is now broken, actually, you’re going to have to apologize to one of the customers and say, “Sorry, it’s no longer in stock.”

You already need this apology workflow in a business in many circumstances anyway. Dealing with this potential serialization violation from accidentally selling the final item in stock to two different people, that actually precisely the same to the user as the forklift truck running over an item in the warehouse case to the end user. Those are things are indistinguishable.

I would argue there that this case where you really, really require an up-to-date read is actually often more malleable than we might think. What we need is integrity of the data, that is we need to make sure that if something was written, it won't be lost, and we need to make sure that the same writes that appear in one data store also appear in another. We can't be losing data randomly. That's sure for truth.

You need the integrity of the data, but you don't necessarily need the timeliness in all cases, the guarantee that some value is 100% up-to-date. I think quite a lot of cases can tolerate a value being a little bit stale, and if you're willing to tolerate that, it actually opens a whole lot of new architectural possibilities, like the classic view of serializability simply doesn't cater for.

[01:01:39.7] JM: I know we're up against time. I would like to ask you one more question. Do you have a hard stop right now, or I can ask you one more question?

[01:01:46.0] No. Go ahead.

[01:01:46.1] JM: Okay. All right. I think about – If we were doing this podcast five years ago, the conversation we would be having would be pretty similar to the conversation that we're having right now. What's interesting about lower level technology discussions as supposed to discussions about consumer technology is that it seems like the predictions that we can make are a little easier to make. If we're doing a show about consumer electronics five years ago, probably, we wouldn't have been able to realize the importance of video, or maybe we would. I don't know – Video sharing.

Anyway, five years ago, we would have been able to talk about things like, "Okay, Redis and Kafka are getting big. Cassandra is going to be bigger. Kafka is going to be bigger in the future. These things are getting really important. Maybe we wouldn't have been able to predict containerization. We might not have been able to predict how highl-level the cloud services are going to get. Clearly, cloud services are just going to get higher level and higher level, and application level programmers are going to be able to deal with these higher level APIs.

Luckily, they are going to have to not think about the slippery transaction problem as much if they want to do, unless they are Netflix, or Uber, or one of these companies that is at such a scale where they have to think about these things a lot more. These out of the box solutions are going to get better. What's going to change? What are the things that are going to change in five years, or 10 years? What's in the future of these data-intensive systems?

[01:03:29.7] We saw in the last 10 years, I guess, quite an explosion of different tools available. As you said, many of the tools we have now were already around five years ago, just less mature. In the five years since then, I guess we've mostly matured those tools and learned a bit on how to use them and how to plug them together and what circumstances to use which tool –

[01:03:53.7] JM: New tools in the same paradigm.

[01:03:55.9] Yeah. Though, I think going forward, what I would expect actually is still occasionally getting new tools, but more importantly better understanding of the abstractions for pulling them together. I could imagine that at some point we reach the kind of patterns for using data systems that might then be like the new equivalent of the relational model.

As I said at the beginning, the relational model really dominated data management world for decades because it really figured out a good way of exposing data to application developers that worked for most of the things that people wanted to do. Now, we've grown beyond it, but I expect that we'll still be aiming for finding that right abstraction that allows us to express the things that we want to do in a nice way without having to worry too much about the details of exactly how it's implemented.

I could well imagine something like that coming along some kind of model of data management which may – It will definitely have to accommodate things like real-time data use and representing the same data in multiple different ways like we discussed, like full-text indexing, and caching, and materialized views, and column store, and row store. It's going to have to somehow accommodate all of those.

What my vision there is, really it would be great if we had something like Unix pipes, but for data systems. Why can't I just say MySQL pipe elastic search and just write that and have it be done.

What that would mean is that the contents of MySQL database are reflected in elastic search and searchable there. Whenever something changes in the MySQL database, then it gets updated accordingly in the full-text search index. That's really ought to be a trivial step, and right now it's not trivial at all. You have to do a huge amount of plumbing to make something like that work.

That's kind of my hope for the next few years, is that we get towards a point where we can just plug these different technologies together better, because we figured out the right abstractions for making them in to operate in such a way that we still keep the advantages of all these different technologies that are optimized to different purposes, but also we get rid of some of the low-level plumbing that we currently have to do just in order to plug these technologies together.

[01:06:31.4] JM: Right. It makes me think again of GraphQL and the middleware flexibility you get there. It also makes me think about something like Google Dataflow. You talk to the Google Dataflow people and they talk a lot about the false dichotomy between batch and streaming. You don't want to be thinking about batch versus streaming. You just want to be thinking about, "Am I getting this data?"

They have to do some really hard conceptual thinking. The stuff that they talk about in their presentation is that Google Dataflow people are – They have some really interesting ideas around how should you reason about batch versus streaming and windowing. That stuff feels like we're – Maybe Google has already got this figured out and they're just gradually unveiling it. In any case, it feels like we're just getting started in how to think about these time windows for different datasets.

[01:07:33.0] MK: Yeah. The Dataflow team is doing some really great thinking there, and they're doing that in the domain of analytics. I think they are not really caring very much about how you would build a transaction processing system simply because that's not what they're there for. They're for building analytic systems, which is fine. That's simply their domain.

Then, if you take GraphQL on the other side, that's very much designed for building Facebook [inaudible 1:08:02.3]. That is you're got a mobile app which requests some data from a server and then displays that data. As far as I know, for example, GraphQL doesn't really have a story

for pushing real-time updates down to the client. The only way how our client would find out about an update to some data is just by pulling repeatedly.

We've got, on the one hand, the Dataflow model which is all about streaming and being able to reflect changes to data in real-time with low latency and treating that in the same way as batch. Then, you've got GraphQL, which kind of ignores the whole real-time thing.

Then, on the other hand, they focus on doing the kind of transaction processing type thing. I could imagine there's space for some kind of unifying abstraction there which somehow takes the best from all of these worlds, but I don't think anyone has come up with it. I think all of these projects are working towards it. I think it's going to be an exciting few years head.

[01:09:02.6] JM: Absolutely. I think on the – Not to draw this conversation even further, but I think on the user application side, the self-driving stuff is just going to push the necessity for bigger, more – Because it's like the systems that we've been building, it's like – Okay, Facebook. You need to serve Facebook mobile traffic. That's a hard problem, but it's not as sensitive as getting your machine learning models up-to-date because your car could potentially drive you off a cliff if you don't have the up-to-date data and the machine learning. I don't know. I think necessity will drive invention.

[01:09:40.9] MK: Yeah, I imagine it so. Let's see what inventions people come up with.

[01:09:41.9] JM: Right. Okay, Martin. It's been a pleasure talking to you. If you ever have another book, or presentation, or something that you want to get some publicity or talking around – You are a very requested guest. Multiple people wrote into me requesting your presence on the podcast. You did not disappointment.

[01:10:04.5] MK: Well, thank you very much.

[END OF INTERVIEW]

[01:10:07.4] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily.

Thanks again Symphono for being a sponsor of Software Engineering Daily for almost a year now. Your continued support allows us to deliver this content to the listeners on a regular basis.

[END]