EPISODE 863

[INTRODUCTION]

[00:00:00] JM: FoundationDB is a multi-model distributed key-value store. It is fully ACID compliant and horizontally scalable. FoundationDB is not usually used directly by an application developer. FoundationDB is a foundational building block for higher-level distributed systems, such as the metadata storage system for a data warehousing tool like Snowflake.

Ryan Worl is a software engineer who specializes in FoundationDB. He joins the show to discuss the architecture of FoundationDB including the roles of different server components and the read and write path of FoundationDB. We also talk about applications of FoundationDB and how it compares to storage engines such as RocksDB and databases such as CockroachDB and Spanner.

If you want to find all of our episodes about distributed systems and large-scale databases, you can check out the Software Daily app for iOS. It includes all 1,000 of our old episodes as well as related links, and greatest hits, and topics, and reading material. You can comment on episodes. You can have discussions with members of the community and you can become a paid subscriber. You can get ad-free episodes of Software Engineering Daily by going to softwareengineeringdaily.com/subscribe.

If you're looking for help with mobile and web development, I recommend checking Altalogy. They're the company that has helped us build the newest version of the Software Daily app for iOS, and the Android app is on its way. It's another 2 to 4 weeks away, and I'm really excited to have two awesome apps for Software Engineering Daily that will soon be in the App Store. The one for iOS is already there and it's quite good. So please check it out if you're interested.

[SPONSOR MESSAGE]

[00:02:02] JM: DigitalOcean is a simple, developer friendly cloud platform. DigitalOcean is optimized to make managing and scaling applications easy with an intuitive API, multiple storage options, integrated firewalls, load balancers and more. With predictable pricing and

flexible configurations and world-class customer support, you'll get access to all the infrastructure services you need to grow. DigitalOcean is simple. If you don't need the complexity of the complex cloud providers, try out DigitalOcean with their simple interface and their great customer support, plus they've got 2,000+ tutorials to help you stay up-to-date with the latest open source software and languages and frameworks. You can get started on DigitalOcean for free at do.co/sedaily.

One thing that makes DigitalOcean special is they're really interested in long-term developer productivity, and I remember one particular example of this when I found a tutorial in DigitalOcean about how to get started on a different cloud provider. I thought that really stood for a sense of confidence, and an attention to just getting developers off the ground faster, and they've continued to do that with DigitalOcean today. All their services are easy to use and have simple interfaces.

Try it out at do.co/sedaily. That's the D-O.C-O/sedaily. You will get started for free with some free credits. Thanks to DigitalOcean for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:04:02] JM: Ryan Worl, welcome to the show.

[00:04:04] RW: Nice to be here, Jeff. Thank you.

[00:04:05] JM: We're going to talk about FoundationDB today, and I'd like to start by discussing the applications of FoundationDB. What would I use FoundationDB to build?

[00:04:16] RW: That is a very tough question, because it's so applicable to so many different problem domains and you'd be able to use it in so many different ways. But just the most basic use case for it would be you need some key-value storage that you need to be able to transact across any of the keys at any time. So, it presents itself as a single system to you and you can do transactions over any keys that you want, but it's really important to know that you're not designed to use it just in that way. You're supposed to build stuff on top that takes advantage of that to do fancier things.

[00:04:52] JM: Is that where the name FoundationDB comes from, the fact that it is meant to be used as a foundation to build more abstract systems?

[00:05:00] RW: Yes, as far as I know, that is the conceit there where the name comes from. The company, it would be 10-years-old today, or not today, but this year if it were still a private company, and that's the origin story as far as I know. But that was a long time ago, and I wasn't there.

[00:05:19] JM: Indeed. So, there are many different key-value stores in the world. People who are listening to this are probably familiar with in-memory systems like Redis. Well, it's not entirely in-memory, or systems like MongoDB. It's a document storage system. There're things like Spanner and CockroachDB. Well, I guess those are more SQL transactionality. You've got RocksDB and maybe we can go through these comparisons a little bit later. But before we get to the comparisons, can you give a more narrow definition of when and why I would use FoundationDB even as a building block for a different system?

[00:05:59] RW: Yeah. So the important criteria you would need to evaluate whether or not FoundationDB is good for your specific use case is if you are expecting the storage to exceed the capacity of a single machine. That would be a good place to start, because it's a distributed key-value store designed to run on multiple machines.

You'd also want to know that you need transactions. There are certain use cases, like an obvious one is purely read-only data where transactions will not be particularly useful. But once you get past those two things, basically, you need something that has the capacity to grow beyond a single machine and you need transactions. Those would be about the two things I would say. There are specific different issues when it comes to performance and the cost of using FoundationDB in terms of maybe it's not as efficient as some other system that you could compare it against. But those more in the weeds, and I think that if you know that you need both scale out storage and a number of requests per second that you can process and you need transactions, those would be two good criteria to start with.

[00:07:08] JM: All right. Well, it's clear to me at this point that we're going to need to delve into the subtleties of FoundationDB in order to truly understand when and why we should be using it. But just so we can anchor our understanding around some prototypical use case, can you tell me one or two places where you have seen FoundationDB would be particularly useful. I don't know if it's a shopping cart application, or a videogame, data management, just some concrete application for us to anchor our understanding around.

[00:07:43] JM: Yeah. So, I think a good example that has been talked about relatively publicly, including at the FoundationDB Summit, is the use case from Snowflake, which is the cloud data warehouse company. They use FoundationDB to manage the metadata for their cloud data warehouse. By metadata, I basically mean – This is a bit in the weeds of their specific architecture. But they store the data for the tables and the data warehouse in S3, and the metadata about what files belonged to what table is in FoundationDB.

Alongside that, they also store all of the kind of generic OLTP type data that people are familiar with, like user records and accounts and permissions and things like that. Those are all stored in FoundationDB as well. So it's a combination of the metadata for the actual data warehouse itself along with all of the supporting cloud services on top that are basically access control and management, things like that.

[00:08:44] JM: Wow! Now I'm seeing Snowflake in a totally different light. We did a show about Snowflake a while ago, and if I recall, one thing that makes Snowflake pretty good, and this is something you want out of a data warehouse, is you've got your piles and piles of data and you want fast access to that data. One way to get fast access to your data is to index it in a bunch of different ways. An index gives you faster access, faster lookup time. I think an index is kind of one form of metadata, data about the data. Like the data that you're actually indexing and building metadata around is the data that you're throwing in your data warehouse, but you need a way to comprehend that data, and that is the metadata store that will exist in a data warehouse. So it makes sense to me that a fast metadata store that is eventually – I think there's kind of an eventual consistency element to FoundationDB. It's not maybe a strongly consistent as –

[00:09:53] RW: Oh, no. It is.

[00:09:55] JM: Oh, it is?

[00:09:55] RW: Yes, it provides the exact same semantics as Spanner does. They're externally consistent, strictly serializable.

[00:10:03] JM: Okay. Then I will stop talking. But I just want to say, it is interesting to see that use case. Could you delve into a little more detail as to why that is a useful application for FoundationDB? I think Snowflake was started – What? Like probably around 9 or 10 years, 8 or 9 years ago, something around the same time as FoundationDB. What were the other options that snowflake as a data warehouse company could've used around that time and why was FoundationDB a useful fit?

[00:10:30] RW: Yeah. So from my understanding, they originally tried to use MySQL, and the reason that they would need a metadata store besides just managing the set of files that are part of the table is they also need – The transactional model for Snowflake is actually snapshot isolation. So as a part of that, you do need to manage ongoing transactions. It's not purely read-only data. As a part of that, they use FoundationDB to manage the locks and other things that quickly – Like they implement the transactions at the FoundationDB layer.

When they were using MySQL – Again, this is from my understanding. I've never worked at Snowflake. They just weren't confident that they would be able to deploy and scale MySQL at the level that they needed and have the confidence that they needed in it, because, for example, in the default replication configuration, replication in MySQL is asynchronous. So, you can lose some data in between a primary and a replica if the primary fails before it's replicated data to the replica.

So, with FoundationDB, the replication is synchronous, and like you're just – You get more guarantees out of it basically than what you get out of MySQL. So that's why it was important for them to have not only a scalable metadata store but something strongly consistent so that they could implement the transaction model for Snowflake.

[00:11:47] JM: And just to build a little bit more about foundational clarification. Again, I think one thing that makes this an abstraction that is useful for building higher-level abstractions is, as I understand, the API for FoundationDB is not something that you would want to give to an application developer, like a higher-level application developer. If I'm building an accounting software, for example, I might want a SQL API or I might want a document database API, I think FoundationDB provides an API that's more granular or more lower level. You wouldn't actually want the accounting software application developer to be accessing FoundationDB direct. Is that accurate? Is the API maybe more for kind of infrastructure developers?

[00:12:35] RW: That is correct. It is purely an API. I mean, I wouldn't want to say you would never want an application developer to use it, because I'm sure for simpler use cases it may be perfectly fine. But for a more complicated app, like accounting software from your example, you'd be implementing a lot of things that are a part of a higher-level database software. For example, secondary indexes.

Secondary indexes are a construct that you build out of keys and values in FoundationDB. Specifically, you just make like one range of keys be a secondary index on a different range of keys, and we can talk about that type of thing more a bit later. But, yes, the things that you would build with it are typically higher-level abstractions for other people to use. Like, for example, Apple open sourced recently the Record Layer, which is a part of – Its use within iCloud is a part of CloudKit, and that is a library exposed to application developers at Apple working on the iCloud system so that they can get the benefits of something that looks roughly like a relational database, although it's not SQL, and it uses FoundationDB for the underlying data storage.

[SPONSOR MESSAGE]

[00:13:59] JM: Cox Automotive is the technology company behind Kelly Blue Book, autotrader.com and many other car sales and information platforms. Cox Automotive transforms the way that the world buys, sells and owns cars. They have the data and the user base to understand the future of car purchasing and ownership.

Cox automotive is looking for software engineers, data engineers, Scrum masters and a variety of other positions to help push the technology forward. If you want to innovate in the world of car buying, selling and ownership, check out coxautotech.com. That's C-O-X-A-U-T-O-T-E-C-H.com to find out more about career opportunities and what it's like working at Cox Automotive.

Cox Automotive isn't a car company. They're a technology company that's transforming the automotive industry.

Thanks to Cox Automotive, and if you want to support the show and check out the job opportunities at Cox Automotive, go to coxautotech.com.

[INTERVIEW CONTINUED]

[00:15:19] JM: Let's talk about the architecture and then we'll get into the transaction system of FoundationDB. The reason I want to discuss those two elements is because I think this makes it more – It's going to be impossible to cover all of the intricacies of FoundationDB. But in order to give people an understanding of how this thing relates to other database systems, I think going through the architecture and the transactionality will be useful.

So, a FoundationDB server cluster, it's a distributed system. It's got three types of servers. You've got the storage nodes, which store the actual data. You've got the coordinators, which run Paxos and do leader election. Then you have a transactional authority. Explain what a transactional authority is.

[00:16:07] RW: Yeah. So in FoundationDB, the transactional authority, it's actually multiple pieces that scale out across multiple machines, but it's kind of its own component that you can think of it that way. What the transactional authority does is it's responsible for – And we'll talk about this more in detail later, but the core, basically thing, that everything is built on top of in FoundationDB is called the version, and the version is just a number. It's a counter.

If people are familiar with database theory, it's basically what you would call a timestamp Oracle and it is just a number that counts up. Goes off about a million times a second, and this is used as a clock within the cluster to order transactions. That's held on a component called the master,

and this is a singleton process on the cluster. I can talk about later why this is not a scalability bottleneck, but just for now, moving on to the other piece, the second most important piece would be the resolvers. The resolvers run an algorithm in parallel across multiple machines or multiple cores, and the resolvers take in transactions broken up by ranges of keys that have been written.

So, FoundationDB is an ordered database. The keys are stored in order, like [inaudible 00:17:25]graphically, and the resolvers detect if transactions conflict between the read version of your transaction and the commit version of your transaction. Just going back to the version number I talked about before. You get one when you start a transaction and you get one when you commit.

So, the resolvers detect if any key has been modified that you read between your read version and your commit version. That's basically what the resolvers do. They do that in parallel, sharded by key range. When one of them says that your transaction has conflicted, a message is sent to the proxy, which is a different role that's kind of part of the transaction system, kind of part of the clients. It's basically the –It's called the proxy for a reason. It intermediates the communication between the client and the rest of the system.

When any one of the resolvers says that your transaction has failed, it will send a message back to the proxy telling that will tell your client to eventually retry your transaction, because FoundationDB is an optimistic concurrency system and you have to be prepared for your transactions failing because of conflicts. The client libraries basically hold your hand in writing a transaction that is easy to retry. They basically all just accept the function that the library itself will call multiple times as your transaction conflicts. So, it's kind of transparent to you. You don't have to think about it very much.

But, yeah, that's basically what the transactional authority does. I think that documentation came back from when it was a commercial product. So it wasn't super detailed, but that's what the underlying implementation looks like from a high-level.

[00:18:56] JM: How many nodes do you need to have a transactional authority function as it's supposed to?

[00:19:02] RW: So, the way that FoundationDB is written, you actually – For production-ready cluster, you need more than this. But all of the roles actually run within a single process, and that is useful for when you're just deploying it on your laptop, for example. But even in a production scale deployment where you may want to have at least five machines in the cluster so that the coordinators can have a quorum of five members to handle failures of any two members without losing the database, there's an indirection in the code between the roles that a physical operating system process performs and the roles of the database. So, it's kind of up to you in terms of how many processes you want to run. But the way that you deploy it is basically you deploy one process per core on the machine and the cluster can handle assigning roles to the different processes, but you also are given some hooks to say, "I would preferred this machine to have this number of processes of this type and this number of processes of that type." So you get a lot of freedom in that configuration. That's where some people stumble with FoundationDB unfortunately, just doing that configuration part. But there're lots of help on the forums.
So, to answer the question. Basically, it's up to you. You need a minimum of three to do a redundant deployment. Five would be more realistic, and that's just five core, but you would obviously use all of the cores on whatever machines you're deploying to. So, that's basically it.

[00:20:39] JM: I've seen database architectures that do not have a component called a transactional authority. Why do I need this thing? What problems does a transactional authority solve?

[00:20:50] RW: The transactional authority in FoundationDB is used because FoundationDB is not architected like a traditional clustered database. I would say looks much more similar to how you would design a multicore database, because, basically, assuming that all of the communication between threads on a machine is reliable, which generally speaking is. You would design a parallel database, as in parallel on a single machine multiple threads. You design it somewhat like this in the sense that you'd have different threads performing the transaction isolation function. You'd have a thread that keeps track of the clock in the machine to basically order the transactions across the multiple threads.

What other databases typically do is they use distributed consensus algorithms, like Paxos or Raft on the commit path of transactions. So they themselves are performing some of the job of isolation and then they also have locks that look much more like a traditional, like a two-phase locking architecture. There're other architectures. But the way that FoundationDB works in general is there is a – The cluster is assumed to be in an available working state. The processes of the transactional authority, that is.

Whenever any of those processes fail, that is detected, and all of them are replaced to basically make a new version of the transaction system. So, its operating all as a cohesive unit, and that is really important for how FoundationDB guarantees its isolation level. There are no transactions will proceed during any failure of the transactional authority, and that sounds kind of scary. You're like, "Oh, is that a single point of failure?" Well, in some sense it is. But in practice, what happens when one of those processes fail that's part of the transactional authority, is the database goes through what's called a recovery, which takes a few seconds to recruit a new process to replace the one that failed. It's really not that big of a deal. Obviously, you don't want it to happen all the time, and it can be slightly disruptive for those two seconds.

But the important thing to know is that the database still maintains all of its guarantees that it's giving to you, like the fact that you have an ACIT transaction and it's not ACIT in the sense of, "Oh, we don't have any of the isolation bugs that are actually named in the SQL spec that can happen during, for example, like repeatable read or read committed." It's like actually externally consistent, strictly serializable in the strongest sense of the word.

[00:23:30] JM: Let's talk through transactions. Let's talk through a read and a write. When a client is interacting with a FoundationDB cluster for a read, so let's say my – For example, my client – Guys, this is a complicated example. But if Snowflake metadata server makes a request to FoundationDB to read some information about the metadata that's in your Snowflake DB, actuals S3 stored data. You've got to do a read. So, this client is going to interact with the transactional authority initially. What happens in that interaction between the client and the transactional authority?

[00:24:17] RW: So for a read transaction, you start a transaction and you talk to the proxy, and there is no non-transactional read. Just so you know, I have to go through this bit in the

beginning here. This happens for any transaction you perform with some caveats that I can talk about later, but that are not that important. They're just performance automizations. Your client will ask the proxy to get you the read version for your transaction.

[00:24:45] JM: By the way, where does the proxy sit? Is that kind of on the client side, or on the server side, or is it middleware?

[00:24:49] RW: The proxy is its own role deployed within the cluster, just like any of the other roles. So, you deploy storage processes, transaction processes, coordinators. One of the roles is just called proxy, and it's just like any of the other roles on the data base. You don't deploy it any differently.

Your client library, which the client library for FoundationDB is it's written in – It's exposed as a C library, and all of the clients make wrappers for the C library. Your client library does very important things for you, and there's a reason why people don't just go implement the protocol and why they use the client library. It's because the client library does a lot of sophisticated things.

One of the things that it does is it batches the requests to proxies to start transactions among multiple transactions on your client. So, you have that first level batching that happens where multiple begin transaction requests are batched together. Then they're sent to the proxy. The proxy does another layer of batching there between other start transaction requests. Then the batch that is sitting on the proxy, the proxy sends a very small message that basically says, "Give me commit versions or give me read versions," and the read version is the highest version that has ever been committed to the database previously, and this is important for maintaining causal consistency among your reads.

When the master sends back the read version to the proxy, the proxy then replies to all of the batches that it has received with the read version for that transaction. So, that is part of the explanation why the master is not conceivably a bottleneck until you're at an absurdly high transaction start rate. So that's getting a read version, because FoundationDB is a multi-version system and optimistic concurrency after theat. For all of the reads after that, your client does not go through the proxy. Your client talks directly to the storage servers that have the data, and this

is another important difference between many other database systems and FoundationDB is that all of the storage replicas of your data can participate equally in reads.

Usually, in other system, you can only read either – If you want the most up-to-date data, you can only read from the leader of that shard. In FoundationDB, it doesn't work that way. You can read from any of the replicas of that shard. By the way, the sharding is not exposed to you. It's all transparent. So your client just talks directly to the source server that has the data and says, "Hi. I would like to read this key at this version, and your client just returns the data to you directly." So any further reads after that, that's all you're doing, so directly talking to storage servers. There's no going through proxies. So it's very efficient.

[00:27:39] JM: When the client gets this read – The read is called a consistent read version. Could you talk a little bit more about the versioning here? What is a read version?

[00:27:54] RW: The read version is just a 64-bit number. Counts up over time, approximately 1 million per second, and the owner of that number is the master. It basically acts as the thing that hands out the versions, and that's important because there can only be one in the cluster if you want to be able to compare these versions and know that there are two parallel tracks of transactions committing in the database, like a split brain scenario basically is what they will be called. There's only one of these.

The version is used in the true sense of a multi-version storage system. So there are different – At different points in time, which are different versions. There could be different values for keys. If you update a key at say version 10 to some value. It doesn't matter what the value is, and a client comes along and says, "I would like to read at version 10." You'll get that value.

If at some point later another client updates that key and say that's at version 20. The client that's stuck at version 10, it will always get the data that's at version 10, or if there's another client that is reading at version 10, it will get the data for version 10. So there are different values for keys at different points in time, and that's how you get a consistent snapshot of the database at that point in time, is all of your reads are done at that version.

[00:29:15] JM: Are there situations in FoundationDB that can lead to problematic read inconsistency, or maybe you could just talk about what – I mean, you spoke about strong consistency earlier. What is the consistency story of FoundationDB? Are there circumstances where maybe I'm trading off – I get a slower database experience in exchange for strong consistency? Tell me about the kind of problematic cases for the read path.

[00:29:45] RW: So, generally speaking, no. You're not trading off especially for reads. You're not trading off anything for consistency. If anything, it's on the write path, where things get more complicated. The way the transactions work in FoundationDB in terms of the consistency model, and here I'm talking about consistency in CAP, in the CAP theorem sense. From that perspective, they're talking about linearizable, which is basically if you have some object and you write to it all. All of the write and reads to that object are happening in what is basically a real-time order. There is no drifting backward and forward in time as different clients to their read. Everybody sees the same thing, and that is a transaction model, and that's linearizability in the CAP theorem sense.

There's also serializeability, and that is the database, or, generally, people are more familiar with it from SQL databases. Serializability is basically just the notion that your transactions happen in a defined order as if one of them executed at a time. It is importantly not a real-time order in the general sense of serializeability. Your transactions can be reordered under serializability. What FoundationDB provides is stronger than both of those, or I should say as strong as both of those combined. You get what is called strict serializability, which is when linearizability plus serializability.

[00:31:13] JM: Okay, let's talk about the write path in more detail. What happens during a write?

[00:31:18] RW: So, in the in the most boring sense, nothing happens during a write. The client library buffers all of your writes locally until you commit your transaction. So, you can be going along, doing whatever you're doing in your transaction, reading and writing, and your writes don't have any latency. Your writes just instantly return to you. The actual write operation where you're calling set on a key, for example, or delete. Both of those operations happen instantly.

When you commit your transaction, your transactions – What are called mutations, which is basically just writes, are all bundled together and they're sent to the proxy, and this is where you basically start the transaction, the transactional authority that you're talking about. This is the start of that.

You're at the beginning when you contact the proxy. You're sending your mutations and what are called read and write conflict ranges, and those are the keys that you read and wrote in your transaction. This is going to sound overly complicated for just doing a simple write, but there are no simple writes in FoundationDB. Every transaction that writes data goes through this process.

So you send your read and write conflict ranges, which are the keys that you read and wrote during your transaction plus the mutations to the proxy. The proxy takes the read and write conflict ranges. Those get sent to the resolver. The resolvers perform the conflict detection that I talked about earlier. If that passes, your mutations get sent over to a different process class that we haven't talked about yet called the transaction logs. Your transaction needs to commit to all of the relevant transaction logs. Importantly, all, not a quorum. By relevant, I mean there is – From what I understand, this is some of my weaker understanding of FoundationDB, is that the way the transaction logs work, is that they are sharded by key range. If you happen to write all of your – If all of the keys that you write happen to be destined for one transaction log, because let's just say that you wrote one key. It's only naturally going to be destined for one shard at that point.

So it needs to be replicated on to three transaction logs to maintain fault-tolerance. This is important. When I say not a quorum, I mean, all three. If your transaction only writes to one shard, that needs to be additionally replicated on to more transaction logs that are for different shards, and you don't have to think about this at all. It's all handled for you. But, basically, your transaction gets replicated on to the right number of transaction logs to maintain fault-tolerance.

Asynchronously, the storage servers all pull the data from the transaction log that owns the shard that they're part of. Storage servers can own multiple shards. The shards in FoundationDB are kept relatively small for even data distribution.

Once your transaction is successfully committed to all of the relevant transaction logs, the proxies will receive a reply saying, “Good to go. Transaction is committed. It passed conflict detection and it's on the transaction logs,” and then it can reply to your client saying the transaction committed successfully.

[00:34:19] JM: Just to recap. When is a write considered fully accepted by the cluster? When is the situation such that I've initiated a write. The write has been fully accepted by the cluster and any read to the cluster will now read the data from that write.

[00:34:36] RW: The point at which your transaction is considered accepted is when you receive a reply saying your transaction committed successfully. But the point at which that actually happens in the database is when it's committed on to all of the transaction logs. There is an important caveat, which is in any database, not just FoundationDB, if the message saying that your transaction committed successfully never arrives at your client, you have no way of knowing whether your transaction committed successfully or not, and that's exposed to you in FoundationDB through an error message called commit unknown result, and that basically just says, “We don't know what happened.” It could be that the network went down right at the inopportune moment when you are about to receive a reply, or it could be that a recovery happened at some point when you're trying to commit your transaction. At that point, you just retry your transaction. The important thing to know is that just like in any other database system, you need to make your transactions item potent.

[00:35:33] JM: Okay. We've explored the transactionality in some detail. We’ve explored the architecture in some detail. I know this is an imperfect way to assess databases, but I'd like to do some comparisons. So, first of all, when I think about lower-level systems that we can build data systems on top of, I think of an abstraction known as a storage engine.

In MongoDB, for example, there's a storage engine called WiredTiger, I think. What's the MySQL? So, MySQL storage engine – Whatever. They have some MySQL storage engine.

[00:36:14] RW: There’s InnoDB, and then there’s also MyRocks, which is RocksDB. That is a newer storage, and that uses RocksDB.

[00:36:23] JM: Right. RocksDB. Okay. Do you consider FoundationDB a storage engine?

[00:36:28] RW: Yes. I would say I do consider FoundationDB a storage engine, but it also has storage engines internally. There are two of them at this point. There is the SSD storage engine, which is it uses the B-tree from SQLite, and then there's also an in-memory option, which, importantly, durably stores the data on disk and it has all of the same transactional guarantees as the SSD storage engine. You just must have data that fits in-memory.

Yes, I would consider a storage engine in the sense of if you swapped a FoundationDB cluster for an embedded key value store that you would consider a storage engine, like RocksDB, for example. The programming model would be very similar.

[00:37:12] JM: Okay. So how would we compare FoundationDB to RocksDB?

[00:37:17] RW: So, the fundamental comparison, I think at the lowest level. If you're using the SSD storage engine in FoundationDB, is that RocksDB is an LSM tree, and there are tradeoffs that are probably outside of the scope of this discussion. But, basically, they are different, and depending on the workload, one can be better than the other. That is very complicated in and of itself.

But the other comparison that you can make is that as far as I know, I am not super familiar with RocksDB, but I don't think the transaction model is as rich. I don't know if RocksDB has multi-versioning, which is an important part of using FoundationDB, is that you can have transactions that are proceeding at multiple versions. So, the comparison I would say is they're actually relatively similar from a programming model perspective. You just need to read the fine print.

[SPONSOR MESSAGE]

[00:38:22] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens, and we don't like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what's the alternative? Triplebyte is the alternative.

[INTERVIEW CONTINUED]

[00:40:42] JM: What about Spanner or Cockroach? These consistent SQL databases, globally consist?

[00:40:50] RW: Yes. So, again, at a relatively superficial level, an important thing is that both of those offer MySQL interface. They also, interestingly enough, are both built on top of key-value stores. If you read the original papers about Spanner and some of the papers further on, like the paper on F1 from Google as well, you'll see that Spanner is a key-value store and they describe it in relatively similar terms to FoundationDB. But FoundationDB is very different from those databases when it comes to how it implements those guarantees.

Both Spanner and CockroachDB attempt to use wall clocks, and Spanner can use wall clocks to maintain its consistency, because Google deploys atomic clocks and GPS clocks in their data center to maintain a very tight bounds on the potential clock skew. You can get that similar system from AWS as well through system called Time Sink, as far as I know, that you could use when you're deploying CockroachDB. CockroachDB uses a different system called hybrid logical clocks, which has its own set of tradeoffs, but both of those databases are attempting to be globally consistent in the sense that you're deploying them in multiple regions, as in multiple largely, separated by hundreds or even thousands of miles, geographically dispersed data centers.

FoundationDB is – Just if you deploy a single cluster in the default configuration, that would not be a usable FoundationDB cluster. It just wouldn't work right. It's not designed to be a globally distributed database. There are newer features that are being added right now, and some of which, which already exist, that allow you to deploy a multi-region cluster, which is essentially multiple clusters that are communicating with each other that allow you to read and write data and you just pay a little bit of a latency penalty from the region that is in the active region.

If you read the fine print of Spanner and CockroachDB, I think that the difference between a multi-region configuration of FoundationDB and the latency penalties that you're paying doing global transactions in Spanner. They're actually not that different. Especially in FoundationDB, if you're okay with stale reads in a multi-region configuration, you can read stale data, very slightly stale, from the secondary region without paying a latency penalty.

So, they're different in the sense that they also try to – They do what I think is best described as partitioned consensus. So, in CockroachDB and in Spanner, they both do consensus on the individual shards of data so that, for example, you could have one shard that's down, that's having issues, and another shard can be committing transactions just fine. If your transactions only touch data that's up, then you don't even notice. But FoundationDB doesn't work that way. If any of the nodes in the transactional subsystem are down, then it has to do a recovery. But as I said, that only takes a couple of seconds.

So, the differences I think are very important, but for an application developer, if you're not deploying a database in multiple regions and you're just, for example, using CockroachDB in

one region, or Spanner in one region, ignoring the SQL aspect of it, which is very important, admittedly, it wouldn't feel that different to you.

[00:44:12] JM: Let's revisit some areas of FoundationDB. FoundationDB has an abstraction called a layer. What is a layer?

[00:44:21] RW: A layer is an agreement between two different pieces of software, potentially just the same software running on multiple machines to agree on the format of keys and values so that keys and values written by one program can be interpreted by another program in some higher-level way that is not just the raw bites of the keys and the values.

I think the best example of this that is basically used by – Its used by most of the higher-level layers that people build is what's called the tuple layer. The tuple layer, it exposes an API that allows you to encode tuples, which in most programming languages are just arrays, and it takes the individual elements of those tuples, like strings, integers, floats, raw bites strings, and it encodes that in a way that will sort intuitively.

So, for example, you could have an array of two values. Could be the string A and the integer one. If you encode that, that will come before. As in if you do a range read, it will come first before another tuple that, for example, could be encoded as first element string A with the integer 2. So you can read things back out in order and you can build these tuples that represent higher-level things in your data model.

If different pieces of software or the same software running on multiple machines all understand the tuple layer and the tuple layer comes with all of the official bindings and probably all the community bindings as well for FoundationDB, you can have different pieces of software interoperating on the same data.

[00:46:07] JM: FoundationDB is written in Flow, which is, as I understand, a language designed for FoundationDB. So this was a domain specific language for creating FoundationDB. It compiles to C++. Why does FoundationDB need its own language?

[00:46:27] RW: Flow is, I would say, mostly C++. Anybody that's programmed in C++ will look at this and say, "This is C++."

What Flow does it say it's a very simple compiler that adds a few features that basically expose actor model concurrency to C++, and it makes it easier to write concurrent programs in C++. The reason why that was necessary is that FoundationDB is tested in a very rigorous way using what's called a deterministic simulation. The reason they needed a new programming language to do this, is that to get a deterministic simulation, you have to make something that is deterministic. It's kind of obvious, but it's hard to do.

For example, if your process interacts with the network, or disks, or clocks, it's not deterministic. If you have multiple threads, not deterministic. So, they needed a way to write a concurrent program that could talk with networks and disks and that type of thing. They needed a way to write a concurrent program that does all of those things that you would think are non-deterministic in a deterministic way.

So, all FoundationDB processes, and FoundationDB, it's basically all written in Flow except a very small amount of it from the SQLite B-tree. The reason why that was useful is that when you use Flow, you get all of these higher level abstraction that let what you do what feels to you like asynchronous stuff, but under the hood, it's all implemented using callbacks in C++, which you can make deterministic by running it in a single thread.

So, there's a scheduler that just calls these callbacks one after another and it's very crazy looking C++ code, like you wouldn't want to read it, but it's because of Flow they were able to implement that deterministic simulation.

[00:48:15] JM: Why did you get interested in FoundationDB?

[00:48:18] RW: The reason that I was interested in FoundationDB is that at many companies that I've worked with and jobs that I've had, there an explosion of different data systems as the company goes from just starting or not that big. They start out with something like a single relational database and then add a background job queue and a cache, and then they add a

search engine, and then they realized that they need to start putting things in Kafka so that it's easier to make those downstream things consistent. It just is piles and piles and piles of stuff.

The way that FoundationDB can help with that is you can build all of these – You could build these systems all inside of FoundationDB with transactions. So it's very trivial to write these big complicated systems that you would need a bunch of different databases for. You can implement the parts of that that you need for yourself in FoundationDB, and it's all in one cluster, and it's just one API for you to learn. It's a lot simpler. It's different, but is very different from the way people are used to thinking about data systems. But I think there are a lot of benefits if you understand it and you buy into the idea that you're going to do some things yourself and it's going to be different, but the product that you end up with at the end is easier to use and better and has stronger guarantees.

[00:49:46] JM: Why did Apple buy FoundationDB?

[00:49:48] RW: That's a great question. I don't actually know a concrete answer, but I think that the reason why anybody would want to use it are very similar to why Apple would buy it. I understand that Apple is a very heavy user of Cassandra, and as you deploy very large Cassandra clusters, as I'm sure people that have used Cassandra would tell you, it can be difficult to deploy large Cassandra clusters that manage a large amount of data. The guarantees you get from Cassandra are not particularly strong. They're nothing like the guarantees you get from FoundationDB, for example.

So I you want to provide – If you have a system like iCloud, for example, which is uses the Record Layer now. You would want your application developers to be able to have – They want to make these systems that store and process a large amount of data, like there's a ton of data in iCloud, but they don't want to have to – They don't want everybody to have to be distributed systems experts to make something that actually works, which is a challenge when you're running at those giant scales, is that if you want to have something usable that actually it scales and it can handle the load of requests that your customers are generating. You need something that – You used to need something like Cassandra that gave up some guarantees in order to get you scalability. But with FoundationDB, it scales fairly well, and I can understand why Apple would want to use it.

[00:51:14] JM: How has the FoundationDB community evolved since that acquisition?

[00:51:20] RW: The FoundationDB community is relatively small, but it's made up – And you wouldn't know this, and I don't think most people know this, but there are a lot of large companies that you have heard of evaluating FoundationDB to use for various use cases in their company. I don't think I have anybody that I'd feel comfortable naming on a show like this. But basically, suffice it to say, even if it doesn't look like the community is big. The community is filled with people that work at large important companies, because FoundationDB is such an interesting and compelling technology that even if doesn't end up being deployed, people get interested enough in it to definitely investigate.

[00:51:59] JM: We've done a number of shows from companies that either explicitly or have been given the qualification from external commentators as this definition of NewSQL. There's this term NewSQL, and I think this is kind of a perspective – It's a multi-model SQL system. There're these systems that they want to solve all the different problems that you want out of a data platform. You want SQL. You want document. You want data warehousing, and maybe even want a data lake. You want as many things as possible out of this same repository of data. That's kind of the way that I see NewSQL. But people have their varying definitions. Do you have a definition for it for NewSQL or do you have a perspective on that term, or that trend, that set up database companies and projects?

[00:53:01] RW: I do. I think that the best definition of SQL that doesn't try to include any new database built in the last 10 years, because it's very easy to do that. I think NewSQL is defined by just a few things. They need a SQL interface. They typically like to piggyback off of some existing wire protocol, like MySQL or PostgreS, because it's easy. Clients are in in every language, and they try to provide a SQL database that will scale to multiple machines without giving up too many of the features that people are used to in other single node relational databases.

Where this gets complicated though is they do give up, most of them, except Spanner and, to an extent, CockroachDB, although I would say the two that have these strongest guarantees would be, that are NewSQL, would be – Spanner would be the strongest, and FoundationDB

has basically the same consistency guarantees as Spanner. They try to provide that it feels like a SQL database, but you can't – And a lot of them you can't turn on serializable isolation, or if you do, it's really slow. But, basically, they're just trying to provide a scale out SQL database. I think that when you get into these other areas that are like documents, you kind of get into – Yeah, I mean, it's a multi-model database would be a better way to describe it than NewSQL. But the database landscape is very complicated now. You have to read the fine print a lot to understand what these systems really do. Even if they offer an API for something, for example, it may not be very good. So you just have to do your research.

[00:54:37] JM: Ryan Worl, thank you for coming on the show. It's been great talking to you.

[00:54:39] RW: Thank you.

[END OF INTERVIEW]

[00:54:43] JM: GoCD is a continuous delivery tool from ThoughtWorks. If you have heard about continuous delivery, but you don't know what it looks like in action, try the GoCD Test Drive at gocd.org/sedaily. GoCD's Test Drive will set up example pipelines for you to see how GoCD manages your continuous delivery workflows. Visualize your deployment pipelines and understand which tests are passing in which tests are failing. Continuous delivery helps you release your software faster and more reliably. Check out GoCD by going to gocd.org/sedaily and try out GoCD to get continuous delivery for your next project.

[END]