EPISODE 1188

[INTRODUCTION]

[00:00:00] JM: GraphQL has changed the common design patterns for the interface between backend and frontend. This is usually achieved by the presence of a GraphQL server, which interprets and federates a query from the frontend to the backend server infrastructure.

Dgraph is a distribute graph database with native GraphQL support. Manish Jain is a founder of Dgraph and joins the show to talk about its purpose and his vision for the future of the technology.

[INTERVIEW]

[00:00:34] JM: Manish, welcome to the show.

[00:00:35] MJ: Thanks for having me, Jeffrey.

[00:00:37] JM: You work on Dgraph, which is a database associated with GraphQL. Most people listening know GraphQL as a protocol, but why does it make sense to think of GraphQL as something to build a database around?

[00:00:50] MJ: Yeah. So our story of using GraphQL for building a database actually starts back in 2015 when I was thinking about building Dgraph. And the graph ecosystem or graph systems ecosystem actually is quite I would say fragmented between different languages. There is no standard like SQL like we have in relational databases. And so you had things like think of Gremlin and Cipher and Metaweb Query Language and a few other things. And before this, I was working at Google and has had a lot of interaction with Metaweb folks, which who Google had acquired that company.

And 2015 when I was thinking about building Dgraph, GraphQL came out and I instantly liked the way it was so much more modern compared to these other graph languages. And so the other thing was that it was very close to Metaweb Query Language that I had some exposure to from my time at Google.

And so I quite liked it just because of the fact that it allowed you to express the query in exactly how you would think about in terms of traversals and joins. For example, you're traversing your friends and that becomes a nested thing in the GraphQL. And then friends of friends would become another nested friend predicate. And the result set was JSON-based, which was an exact representation or very similar to the original query.

There's a bunch of like useful aspects of the language that I felt would be great for us to base our sort of implementation of the data based on. But surely it was not without its caveats, right? Because we started with official GraphQL spec but then ended up forking it into the Dgraph query language, which is essentially a fork of it to add the kind of features that you would need from a database. For example, stuff like having variable assignments in the query and transactions. And we actually have these blocks where you can compute different things in the same network call. Stuff like that which are helpful for doing more complex stuff from a query language and also for getting more performance.

[00:02:56] JM: Now the typical setup for a GraphQL system is you have the GraphQL middleware that fetches data from various other sources that it federates the query to. Is this more like a system where you're consolidating all of the data that GraphQL would be fetching from into one single system?

[00:03:19] MJ: That would be a great thing to do. So I think if you care about performance, if you are committed to GraphQL and you actually want to build a big system with GraphQL, then something like a Dgraph where you can take this data from this multiple datasets or multiple systems and putting that into one single system and then being able to efficiently execute your queries across different data models, across data from different systems, multiple data sources. That's where Dgraph really shines, right?

I think of course the easiest way to get started with GraphQL is to do a federation layer on top of your existing microservices or existing different systems. But the problem that happens there is that at some point you start to realize that you are giving up on the performance by doing joints across different data sources or by doing traverses across data sources. And that's when you start to hear about the N+1 problem or the caching problem or some of the things that people in the GraphQL ecosystem have to deal with.

Now, I actually want to draw parallels, because these are exactly the kind of problems that graph systems have had to deal with because they also suffer from this traversal arbitrary depth joints and how slow things get if you were to touch different systems for the same query. And so these are the kind of problems that my team that we solved back at Google with the graph implementation that we're doing there and then we solved it again at Dgraph. And I think now the GraphQL ecosystem is starting to talk about these problems as well. So there's an easy way to get started with GraphQL, which is to do the federation, and that's great. But then if you're committed, then you probably want that kind of performance that you would want from a graphical system and that's where Dgraph comes in.

[00:05:04] JM: It's very hard to get traction on a new database product. Have you actually gotten users to pick up Dgraph as a database to use for their applications in production?

[00:05:17] MJ: You are right. I think it is hard for a new database product to enter into the market, and I think that's the story that we hear that's not just true just for data, but that's true for other databases that I have talked to. Because essentially you are saying, “Hey, you need to trust us with moving your data from wherever it is right now to this new system.”

So I think where we get a lot of traction is when people are already affected by their existing data sources. So for example, let's say they're using some version of SQL and their data size grows to a certain extent and the complexity of joins that you need to do and the kind of data unions across different data models that they need to do. That's when they start to realize, “You know what? There is a tipping point I think somewhere where SQL becomes unwieldy and data at larger scale and the complexity of queries." And that's when they start to think about, "Hey, it would be good to have a dedicated system doing these kind of like complex operations."

And if you look at like the state of infrastructure instead of like software in general, queries are not getting simpler. Datasets are not getting smaller. Complexity of queries are increasing. Complexity of data relations is increasing. Datasets are becoming bigger. So that's where you start to see a lot of people either adopting a graph solution from the market or having to build one internally. And in terms of the build sort of equation, a lot of companies in the valley I think have had to go build a graph system themselves.

[00:06:47] JM: And that said, of course the rewards for building a database that is widely adopted are tremendous. I mean you may just have to look at companies like MongoDB or, I guess, Oracle back in the day. Are you expecting this to be a database that has really, really wide applicability? I mean when you look at the applications that are being built today, what is it about the queries or the infrastructure, the middleware and the backend and the frontend the relationship between those three that makes you think Dgraph is going to be a widely adopted solution?

[00:07:22] MJ: I feel like at some level if you are into, let's say, GraphQL ecosystem, there is no vertically integrated solution out there, right? So most of the solutions that are out there right now are GraphQL layers on top of other databases. And, again, that makes it easier for adoption and easier to get started. But there is a lifetime to it depending upon the size of the data. And that was same problem that we were seeing in graph ecosystem as well. We had a lot of graph layers on top of other databases, which would not perform and therefore there was a need to build a native graph system.

Now there was another part of the question which I'm blanking on right now. Can you repeat that please?

[00:08:04] JM: Yeah, two things. One was – It was a statement plus a question. The statement was the rewards for getting mass adoption of your database are tremendous. The question was more what is it about the communication patterns between frontend, backend and middleware that make you think that Dgraph can be a widely adopted solution?

[00:08:23] MJ: Yeah. So I think we talked about federation before where we are definitely seeing a lot of adoption for a system where the frontend or the client can talk to one thing, which can then just give you the data for all the different datasets that you might have, right? And I think we generally all agree that that's a good model and that's been a very prominent model in the GraphQL or the graph ecosystem.

And if you were to take that a bit further and say, “You know what? Not only do we want to do those kind of like unions of data models. We actually want to make it performant.” So that just is a natural logical step once you're convinced that you need a solution like that, right? So I think that is what that the ability to unify data, different data models into a single system, that is what we were aiming for with Google's knowledge graph and the OneBox’s that Google had. And the same model that I think was at play with Facebook style, probably the same model around Dropbox's edge store and Airbnb's Knowledge Graph and so and so forth. So we have seen that cycle repeat across different companies because at some point they realized that the data models that they have are becoming quite diverse and they are not as well defined as a typical database would want.

And so having them in one single system allows you to be able to not only optimize your operations on it, but allows you to do more complex queries. And I think that's where I feel something like a Dgraph is gaining a lot of adoption when they want to unify multiple different data sources or multiple different data models.

[00:10:07] JM: And it's kind of ironic that you start off with a system, GraphQL, that wants to reach out to all these different disparate data sources. And then if you come full circle to Dgraph, you get a system where GraphQL is just simply the query language against a single database that backs all of these diverse data types in one place.

[00:10:28] MJ: Right. Yeah. I think what that federation is doing is it's making people realize that there is a better way, right? Like if you were to start thinking about, “Hey, how do a GraphQL ecosystem look let's, say, a couple of years from now?” Then people are convinced that, “You know what? Like we need to have – Microservices are great, but we need to have one way to be able to talk through all of these things, because that just makes it easier for us to manage our infrastructure.” The natural next step is, “Okay, let's get the performance. Let's get the ease of deployment.” Stuff like that. And that kind of leads you to a vertically integrated solution like Dgraph.

[00:11:05] JM: So do you see people if they adopt Dgraph they ETL all their transactional or all their like other transactional data into Dgraph so that they have like a replicated copy of it and just it's more accessible from the Dgraph database? Or do you see people just using Dgraph as the single source of truth? Is it like a mirror of other databases or is it the single source of truth?

[00:11:28] MJ: Yeah. So we have seen both the models. So we have seen places where our clients had data in this big Oracle SQL databases and they took that data and put that into Dgraph and now Dgraph is the source of truth, potentially streaming the data from Dgraph out back to those SQL databases. And there’s one case and other cases where they are still using their relational systems but they stream the data into Dgraph potentially via Kafka so that Dgraph becomes a secondary storage sort of where they can run those complex queries and able to – But it's not a source of truth, but it's a secondary storage allowing you to do more complex operations.

[00:12:11] JM: So I can kind of see the problem that Dgraph can slot into solving. It's like you're in a company where you start to pick up GraphQL. GraphQL starts to be widely used throughout the company, but the GraphQL queries can start to get slow because you have all this federation that needs to go on to query the data from the disparate sources. And then if the GraphQL queries are on mission critical applications like fetching user data for a profile page, then you don't want that to be slow. So maybe in order to increase the first round trip time, you can insert Dgraph to just be a copy of all the data that consumes the queries in the exact same way as the GraphQL layer did previously.

[00:12:56] MJ: Absolutely. Yeah, exactly. And you can make it the secondary copy. But then the baby design Dgraph, we were quite inspired by Spanner. And spanner as you might know is Google's geographically distributed SQL database. And so underneath, if you were to look underneath all of these systems, they are essentially transactional key value databases, right? And that's true for Dgraph. That's true for Spanner and so and so forth.

And then the difference is that the way they store the data or how do they decide how to store the data, the way they shard their data and the query language that they put on top, right? So in an alternative universe, you could take a Dgraph and swap out GraphQL with SQL and it might work just fine. But essentially like the way we built it is we built it with the premise that we want this to be the primary source of truth, to be capable of being the primary source of truth, to allow for the scalability that you need, to allow for consistent replication to avoid the eventual consistency data models, which are more complicated for developers to deal with and distributed transactions and so and so forth. So it's actually like designed to be a primary database and to be a very scalable database. But of course, you can use it in many different ways.

[00:14:13] JM: So let's start to get into the internals a little bit. You're saying you modeled it a little bit after Spanner. So this is a database that is built entirely from the ground up, right? You're not using Postgres under the hood or something like that.

[00:14:27] MJ: No. I think below Dgraph is just disk RAM and network and CPU.

[00:14:34] JM: Okay. And did you know anything about building databases from scratch before you started on Dgraph?

[00:14:40] MJ: I think I learned a bunch about building the graph system. So before Dgraph, I was at Google for six and a half years in total right out of college and worked initially in the web search infrastructure for three years working on Google's real-time indexing systems. I learned quite a bunch about building real time distributed large-scale systems. And then was working on knowledge graph and these early days of Google. So we didn't have any infra around knowledge. So I was one of the tech leads to build the Google's graph indexing and serving system.

So I knew a bunch about how would you build a low latency, high throughput sort of like large scale graph system, but not necessarily from the database aspect of things, but from the serving side aspect of things. So we would keep everything in memory and we would try to optimize for low latencies and stuff. And once I came out of Google and started to look at the graph space, it felt like that the database is the first step to what people need. Because I felt there was not a very sort of reliable open source distributed scalable graph system that was out there.

[00:15:51] JM: So give me a little bit on the nuts and bolts of how you start building a database. It sounds like a really arduous task. Give me some of the early days like architecting the early lower level primitives of Dgraph.

[00:16:07] MJ: Yeah, such a journey. Databases are hard to build, right? And I remember very early days just sitting as me and a machine that I had put together because I needed a big machine to be able to use up the CPUs and test it. And we actually store the data in this way where we actually use the concept from search engines called posting lists. So what we do is the entire graph, we assign these unique integer IDs to each node in the system. So if you need to get your friends, we can do a single lookup to get the unique IDs which represent all of your friends. So these controversies are really efficient.

The reason we designed it like that is because we wanted to make sure that if we need to do an intersection between, let's say, your friends and somebody else's friends, it would just be two seeks and then a really fast intersection. So we use this concept from search engines called posting lists, which are essentially sorted list of integers. And I remember like trying to find a great way to make that work, and that piece of the code that we had, we probably – I traded upon it like three times within that same – Like within a three to six months period. And over the lifetime of Dgraph, we probably have upgraded upon this around at least six times having rewritten it and sort of changed the data storage format and changed the way it would be more efficient and so and so forth. And actually we still have more work to do there about how do we represent it to be more efficient.

So that's one part of the equation. The other part is Badger. Early on we were using RocksDB for data storage and we realized that Go – Because Go is the language that we're using. Go was just not very – It does not interact really well with C++ APIs, and there's a course to it, and it was just hard to switch between Go and C, Go ecosystem, to RocksDB, C++ ecosystem. And so we ended up writing Badger, which is an embeddable key value database, which would be a core replacement for RocksDB. And we spent maybe six months building that and it turns out the way we were storing the data, we ended up being faster than RocksDB in certain cases. And that has taken a life of its own. So Badger now is like 8,000 I think GitHub stars and quite popular in the Go ecosystem.

Similarly we built like a caching system called Ristretto upon something that we had in Java. Not to mention Raft and distributed transactions was a very hard piece to get right particularly with consistent replication. We did Jepsen testing in 2018 and it showed so many issues in our transactional system and that we spend the rest of the year fixing those and improving those and simplifying the system. So one thing we learned is distributed systems are hard and it's just a lot easier to build a single server system than a distributor system. But once distributed system works, it's such a delight to look at.

[00:19:08] JM: Now, I don't actually know why you wouldn't just try to layer this thing on top of Spanner, for example. Why wouldn't you just try to build some like query interpretation layer on top of Spanner and then just go from there?

[00:19:25] MJ: Yeah. So we had that option. We could have used even a SQL, right? We could have used something like a Postgres and had multiple instances of Postgres running and then layer this on top. The problem there that happens is, again, this only affects like heavy users.

Like for small users, they won't see a difference. But heavy users, the problem that happens is the issue of deep traversals and arbitrary depth joins.

In a distributed system, what you want – In a distributed graph system particularly, but in a general distribute system, what you want to do is you want to decrease the number of machines that are touched every query. And the problem about traversals is that each traversal adds more data, right? And so what you then end up doing is if your data is, let's say, sharded by nodes or keys, then what you will end up doing is that as you go from like first result to let's say 100 results, then those 100 results will go from 100 to 1000 results and then you might filter them by age or something and get the 10. So a user only got 10 results at the end, but internally you had to expand to thousands of million results. And that would end up potentially hitting all of the machines in your cluster in your distribute system. And that's really slow, because the more machines you touch, the slower the current latency gets.

And so we wanted to avoid that by really tightly controlling how we do network calls. How many machines we touch and how we do those network calls? And that was only possible by controlling the storage, because then we can make sure that they're not doing any extra work, and by controlling how we store data and how we shard data. And so while it was possible to use some other system to do the storage, ultimately the amount of work that would have been required and the kind of results we would have gotten, it was better to like just build it from scratch and be able to get those kind of performances that we care about.

And I think if you were to – Like I actually asked myself the same question, like why did I not just do a simpler way? And I think knowing what I knew from Google that like having a distributed system, which was built from scratch, was the best way to do it. But if I had not had that experience of scale and all that stuff, then a single server system would have been a very legit way of going about it as well.

[00:21:49] JM: Are there any low-level bugs that you could share from ironing out from the early days?

[00:21:55] MJ: Oh! Yeah, so many bugs. I think particularly, again, with the distributed system issue, right? And Raft is such a complex piece of stuff. And I remember we had built this Raft. So the way Raft works is that, let's say, you have three machines in your Raft cluster and you want to make some changes to the cluster. So you make a proposal and then all of the machines would apply it. All of the machines would achieve a quorum, achieve consensus, and then they'll apply it in the same order. So you can think of Raft as a way to serialize deltas across machines.

And I remember like writing this early code with Raft and we were using this library from etcd and we were still learning about Raft, and we wrote this proposal. The way the code was written was you made the proposal and then you wait for it to show up after it has achieved consensus, which seemed like a very right way to do things, and it worked. Then we decided to like deploy it and then we realized that, "You know what? Sometimes those proposals just never show up." Because Raft would just drop them and there won't be any errors. There would not be any indication that that happened. And once we realized that, then we had to now work away by which we will retry after so often if we don't see it. And that then led us to this whole exponential back off system where we had to retry after some time, but then that goes other issues around, "Hey, now we are retrying. So that means that proposal has to be applied multiple times, which slows down the whole operation." And so as you can see, it becomes like this rabbit hole where you're trying to solve one thing and then you have to solve like five more things, which are caused by the solution. That's what makes distributed systems hard.

[00:23:44] JM: When did you first start to get production users?

[00:23:47] MJ: Because Dgraph was open source from the get-go, we had sort of a bunch of companies who were using Dgraph, the opposite version of Dgraph on their own, and we would interact with them over our community forums and stuff. But the first sort of signed contract with the monetary value, that happened in 2018. That's when we got our first big customer, a fortune 500 company, and they were the ones who had the Oracle SQL databases that they needed to integrate into Dgraph and be able to build their app. And they had an interesting, very interesting use case, which wasn’t quite like what we're doing at Google again, the unification of different data models. Yeah. So that was the first customer in 2018.

[00:24:31] JM: So as time has progressed, have there been use cases of Dgraph that have shown up that have surprised you?

[00:24:37] MJ: Yes, a few of them. So I think what we see some folks do is they – Particularly, Elasticsearch, I think we hear about, because Dgraph supported full text search, general expressions, term indexing and geolocation search, even passwords. We have a way by which you can store passwords more safely.

So Dgraph had all of these things. And a few users who were using Elasticsearch for full text search, they also wanted to be able to use graphs so that they can do more advanced things. So a bunch of them actually came to us to see if they could use Dgraph instead of elastic. And that's something that we keep on hearing about. Even though Elastic is a search engine, it's not by quality – It's not a database. At least, it was originally designed to be a database. Now I think they might have been transiting towards a database. And Dgraph is database. That means like the way we write everything, it has to be first persisted to disk and we need to get an acknowledgement that it was flush to disk.

Before, we are able to say, “Yes, we did that right,” because you cannot have a database which loses data by any means. The database can crash and the network can go down and all of those [inaudible 00:25:51] can move back and forth. All these things can happen, but the database cannot lose data. So the kind of right path that we have is a lot more complex than what you would do if you were to build a search engine.

But yeah, we were quite surprised by that and we still see a lot of people actually like replacing Elastic with Dgraph. That's one. And second one was time series. Because Dgraph actually has date time and it does different kinds of date time indices. We do day-based, minute-based, second-based, hour-base, so and so forth. A bunch of people actually use Dgraph to store, if not the entirety of the time series data, but at least like chunks of their time series data so that they can do those complex graph queries with the time series data available within Dgraph. So these are the two use cases where we were like we felt like this was sort of not what we had originally thought about, but were very pleasant surprises.

[00:26:44] JM: You mentioned this pattern where people stream another database into Dgraph and then stream the data from Dgraph back out into the other database. What's the latency like on that streaming? Is it a big issue for eventual consistency?

[00:27:01] MJ: I think that part, generally, we are not very deeply involved there, because that's something that the users typically do at their end. So they would probably be using a Kafka stream to input data Dgraph and they might be taking sort of exports from Dgraph to put it back into their systems. So officially speaking, we haven't yet launched the change data capture, which is one of the requests that we have, one of the features that we have in the roadmap. Though our GraphQL solution actually has subscriptions by which you can subscribe to a query and get updates from it, that kind of helps. But yeah, because data doesn't have an official way to do this, it's hard for us to comment. It's hard for me to comment. But when we do add that, the way our subscriptions work, and because we already have that code internally, it's pretty instantaneous. So the moment we write to Badger, which is the database that we use for writing to disk, the moment we write to Badger, at the same time we send out signals, we send out the data over to any subscriber who might want to listen to it, which means that the delay should be very minimal.

[00:28:09] JM: Are there any tough lessons you've learned about architecting a database, like mistakes you've made that you have to go in and completely re-architect parts of it?

[00:28:17] MJ: Oh yeah, absolutely. And the distributed transactional system with consistent replication, that was a very hard one for us to get, because we couldn't find a paper on it which would describe that. So first of all, graph systems in general don't have – There is no industry written paper, right? For example, Spanner has a paper written by Google. Then MongoDB and others, they were inspired I think by Bigtable's. And so there was a bunch of papers that you can base your architecture upon. But for graphs, we didn't have that. So that had its own challenges on how we built things. But then distributed transactions with consistent replication, the way we had to devise it, we read papers from EdgeSpace, from Percolator, from other places, but nothing came close to what would fit in our use case. And so we ended up devising a new way to do transactions.

And in doing so, I think the first iterations of it were quite complicated. And because of that, when we did the Jepsen test, we realized that we were seeing these lots of edge cases where transactions were failing. So one thing that we learned while solving that is to make a distributed system run like a clockwork, which means it's very predictable in what it's going to do. If you're seeing a distributed system and you see an event, then you should be able to almost predict what the next events are going to be. And by applying that principle in how we do operations in the database, in how we design it, we were able to really simplify the operational database and also be able to debug for edge cases and avoid those edge cases in the first place. So making the distributed system run like clockwork is a principle that we have and we really try to think about it every time we design something. That's one part of the equation.

[00:30:10] JM: So what about going to market with a database? Any lessons there you've learned?

[00:30:15] MJ: I think with the database, the go to market is, again, I think you need to figure out where the right adoption is, right? And just to give you an example, when Go was written, the original idea for Go was that a bunch of C++ programmers are going to switch over to Go. But what they realized after having launched it is that it was the Python programmers who were switching over to Go, right? It was completely not what they expected.

And so with the database as well, it's important to understand where is the actual pain point where it makes sense for somebody to give up what they have and learning that they might already have and adopt this new solution, which is at such a core of their technology, right? Where if that solution fails, everything above it fails, because database is literally at the bottom of the entire stack. And if the database goes down, nothing else would work, right? So that pain point has to be really clear. And then you can go target those potential use cases and potential customers who actually have their pain points.

So I think one thing that we realize for ourselves is that it is slightly harder for an average sort of like one or two person developer to switch away from Postgres, because they are very used to it and they like it and their friends are on it, so and so forth. But then as they start to grow their datasets and they start to have different data models and they start to do joins, then the complexity of it grows pretty rapidly, because then they need to do data denormalization, copying of data to different tables to avoid the cost of joints and so on and so forth. And that's where we started to like focus on more and see how we could solve those issues.

So each database, if you look in the history, each database had a very specific case that they went after. MongoDB made it easy for people to do JSON. Influx is very specific about time series. Elastic was about search and so and so forth. So I think that use case has to be clear.

[00:32:18] JM: What areas of the database are you focused on working on right now?

[00:32:22] MJ: I think the big one for us right now, so far we have been focused upon scaling Dgraph to the dozens of terabytes of data, and I think we've already achieved that. And I think now, I think the big focus is to scale it to hundreds of terabytes of data, because that's the next big milestone for us. And so that's what we're going to be focusing on next year. But at the same time, there are certain things, some of the features like multi-tenancy, which comes up. Because once you deploy Dgraph in a company, most likely the team sitting next to you also wants to use the system and then they don't want to have to spin up another Dgraph cluster to be able to use that. And so multi-tenancy becomes quite useful.

And also we have cloud services like /GraphQL, which makes it easy for a developer to get a GraphQL backend. We need multi-tenancy there as well so we can have more tenants on one bigger deal of cluster. Because, again, Dgraph can scale to, as mentioned, terabytes of data. There's no point in being enough clusters, tons of data clusters for very, very small datasets. So multi-tenancy is something that that we're looking at in the next year. And then we have these laundry list of features like generator capture, more features in GraphQL itself, audit logs and so and so forth that we are looking at for next year.

[00:33:42] JM: The application of a developer wanting a GraphQL backend from day one, how often is that? How frequent is that? Are people just opting for GraphQL as their main communication pattern these days?

[00:33:55] MJ: I think it's a bit early to say. We definitely see a lot of excitement in the community. And every other day there is a company which is switching to GraphQL, a big company that people respect, and that actually just only drives that excitement even forward. We see, for example, a bunch of users using /GraphQL that we have to build a new app or build a new system that is not just what we call a greenfield project. So there's definitely excitement, but what percentage of developers are choosing that over their typical like REST and SQL route? That's a bit hard to tell right now. I think it might become more clear in maybe a year from now or maybe a couple of years from now. But just given how young GraphQL is it's hard to differentiate between the excitement around the area versus the actual like numbers in these many people are choosing GraphQl over REST APIs when they're starting a new architecture.

[00:34:56] JM: Do you think GraphQl is more intuitive to the new developers that are coming up these days? Like it seems like if I'm a new developer learning these days, I'm usually starting with JavaScript. And what kinds of queries am I going to make? Well, I'm going to make queries that look like JSON. So probably I'm going to be making GraphQl style queries.

[00:35:17] JM: I think it's absolutely a lot more intuitive, because, also, I mean the other part of the equation is that developers are iterating, right? When they are building something, they don't yet know all the things that they would need. There was a time when people would start and create these waterfall diagrams and try to figure out all the things that they need up front, but that's not how we operate in software development today. Today it's all about Agile and you go build that MVP and then you trade up on it and you trade up on it and you trade up on it, right? And so that kind of development system, GraphQL is able to do that very well, right?

So you can go, you can modify and you can differentiate between your backend, which is implementing GraphQL and your frontend, which is consuming GraphQL. You can have sort of like a testing layer in between, which can differentiate that. So your frontend development can be quite separated out from a backend development.

So I think it's very intuitive for a consumer of GraphQL. It is not as intuitive I would say for the developer of the GraphQL backend. I think the developer GraphQL backend, honestly, I feel it would be harder for them to build those APIs, but the consumer, it's going to be extremely convenient for them because they can see the introspection, they can get their query auto completion automatically. They can change their queries to just give me a bit more data or a bit less data and so and so forth. So great for frontend, not as easy for the backend developer.

[00:36:45] JM: Let's go through the life cycle of a transaction on a Dgraph database. So let's say I'm a developer. I issue a query from the frontend. It hits the Dgraph server. What's the life cycle of a query being processed?

[00:37:01] MJ: Yeah, that's a great question. So the way it works is that you can hit – Let's think about a typical architecture. So if you want high availability cluster and data is full tolerant. So if you want high availability cluster, you would have three what we call zeros and three alpha. So zeros are managers and alphas are the data holders. And so you can hit any of the alphas to do your queries. And alphas are distributed among groups. So each group is a replica. All the members of a group are replicas of each other and then different groups are data sharded across groups, right? So group one would have, let's say, half the data. Group two would have half the data. Group three would have – If you had group three, they would all have one-third of the data and so and so forth.

So you can hit any of the alphas for your, in this case, let's say a write, or a query. And so what we do is as you hit one of the alphas, it's going to talk to the zero and get a timestamp. And this is for those instantly synchronous query results. If you want eventual consistency, it can use the timestamp that's already present on the alpha, which will give a bunch eventual consistency. But for the default mode of Dgraph, which is very strict consistency, it's going to go and ask zero for a timestamp. Once it has a timestamp, that is going to figure out how does it need to serve this query? So it's going to figure out which alphas groups have what kind of data. It sort of knows that already. It's going to look at the query and figure out how does it need to divide up that query into smaller tasks. And that is going to shoot out those tasks over to different alpha groups. And part of it, it might be able to serve locally based upon the data that it already has.

So for all of these tasks, they all have the timestamp. So each of the tasks will land on the corresponding alpha. They will look at the time stamp and they're going to see, “Hey, am I up-to-date with this timestamp that I have?” And each of these alphas are actually getting updates from zero about, where zero is at with respect to the timestamp. So you can think of timestamp as a way for each alpha to know like how far ahead am I, right?

So the alpha would look at what time stamp it has and would look at what time stand the query wants. And if what it has is if it's ahead of what the query wants, it's going to service that query. If it is behind that, it's going to wait until it gets ahead and then service that, right? And so that allows us to get the latest data that is in the system also without having too much complexity of metadata transmission. All you need is that timestamp, right? Now that's one way the query would run. And then what will happen is you did one step of the query. Then let's say you need to now go deeper, you need to do more traversals. Good. Query results will be sent back to the original alpha. We will again generate a list of UIDs that we might need to expand on and shoot that next task over to whichever is relevant and get the results back and so and so forth.

What will happen is that we'll divide up this query into many small tasks, which can be executed concurrently. And then once the result comes back, we would recreate the next task that needs to be executed and go for it. Now that's where the interesting part lies in how we solve the N+1 problem, is that if we actually have, let's say, a thousand UIDs which came back from one task. Now we would not bombard like five alphas with those thousand UIDs. Instead we're going to have a bigger payload that will be sent out to the relevant alpha in a single network call. And if that alpha is not responding back within a given set of time, let's say a second, then it's going to shoot that same payload to a replica. So that way we can make sure that even if some alpha has a hiccup, we can still achieve low latency by using the replicas to do the same work. And that's the inside that, by the way, Google uses in their serving system of this. Shooting the query to a replica in case the first one is not responding. So Dgraph does that as well and that that really like works great for us, because at any point of time you can assume that in a real cluster some machine is having a hiccup. You need to keep that into account while you execute the queries to get the lowest latency.

[00:41:22] JM: That's a great walkthrough. So can you explain in a little more detail how data is spread out among different servers in a Dgraph cluster? I mean do you just use the same sharding scheme as a spanner?

[00:41:34] MJ: No. So in other databases, typically you would shard by equivalent of nodes. So if you were to think about a graph, you would say, "Well, I have –" Overall there's like, let's say, five million people in this graph. So I'm going to distribute these five million people or five million entities across the different servers so that each server or each entity then fully contains the data that that entity has. That's how you would typically design it and that's how I see it be designed in bunch of places. But that's not how we do it in Dgraph.

In Dgraph what we do is we would say these are the "predicates" or relationships that an entity can have with other entities or values. And we will have this list of predicates. For example, your name could be a predicate. Your friend is a predicate. Your age is a predicate. Your school, your date of birth, etc. All of these would be predicates. And then we would spread these predicates out across the cluster. Then what that lets us do is that if you just need your friends of friends, then we have exactly one server to go to to get all the friend queries executed. And that becomes very efficient, because if you now need to, let's say, expand from yourself to your, let's say, 100 friends to your friends of friends, which would be hundred, into hundred, let's 10,000, not assuming the similarity between friendships sometimes. You can just do two network calls. The first network call to the friend serving server. Get the 100 result. And then another network call, potentially the same server asking for expansion or traversal of those hundred friends to get their friends. And so in two network course you can execute this entire query versus if you were to do entity-based sharding, then after the 100 result of the first expansion of your friends, you would then have to do almost a network broadcast to all the machines, because the database is randomly sharded to all the machines to get their friend list.

And so it's a very different model where you're going as predicate first or relationship first query execution, but finally, once you have the results internally in Dgraph, you are still converting them back to nodes and giving a node-based structure back to the user, right? So for example, if somebody asked for give me the name and the – For my friends of friends, give me their names and their date of births, right? So we'll go expand the friends of friends and get the list of like 10,000 UIDs. And then we will shoot two different queries, one for expand the name of these 10,000 UIDs and expand the age of the of 10,000 UIDs and get those back and then stitch them together. So then you get your list of friends. The name, the age and the UID would be in one single JSON map, right?

So the user doesn't really see all that chopping up of data and spreading it across, because ultimately what they're getting is a sort of like a JSON document and completely oblivious to how it was stored internally in Dgraph.

[00:44:46] JM: Okay. Well, we're running up against time. Do you have anything you want to add about the future of Dgraph or predictions about database backends in general?

[00:44:56] MJ: Yeah. I think the premise of building a Dgraph was that data models are getting more complicated and the queries that people need to execute. A case is data morals is also getting very complicated. And I think GraphQL is a great testament to that world and it's a great step in that direction. And I feel that you know if you look at purely from a graph system perspective, GraphQL has done great for advancing the state of graph systems. And, initially, GraphQL came out as just an API language. But now if you look in the GraphQL ecosystem, there is so many different players all serving data from different databases and building the GraphQL layer. So that actually gives me a lot of confidence in the way the GraphQL ecosystem’s graph and GraphQL ecosystem are going. And I feel like there will be a point in the future where people might want, when they are building their new systems, instead of like trying to think about how would it look in SQL, they might just start thinking about how it would look in GraphQL and building the systems around that. And so I think that's the future that really excites me for this entire ecosystem and that's the future that we want to build with Dgraph.

[00:46:05] JM: Okay. Well, thanks for coming on the show. It's been a real pleasure talking to you.

[00:46:09] MJ: Thanks for having me.

[END]