EPISODE 1008

[INTRODUCTION]

[00:00:00] JM: Distributed stream processing frameworks are used to rapidly ingest and aggregate large volumes of incoming data. These frameworks often require the application developer to write imperative logic describing how that data should be processed. For example, a high-volume of click stream data that's getting buffered to Kafka needs to have a stream processing system evaluate that data off of Kafka and prepared for a data warehouse, or Spark, or some other queryable environment.

In practice, many developers simply want to have that high-volume of data become queryable in the fewest number of steps possible. The end user usually wants to use declarative SQL syntax to access the data. So why do we have imperative logic along the way? Why do we have these imperative stream processing systems?

Materialize is a streaming SQL materialized view engine that provides materialized views over streaming data. The materialized views are incrementally updated overtime and reconciled with new data that may have come in out of order.

Arjun Narayan and Frank McSherry are the cofounder of Materialize, a company whose technology is based on the Naiad Paper, which was written at Microsoft Research. Arjun and Frank join the show to talk about modern streaming systems and their strategy for taking an academic paper, which was called Naiad, and productizing it.

I hope you enjoy this episode, and if you're looking for any other episodes about data engineering, you can go to softwaredaily.com, you can search, you can find all the episodes about Spark, or other streaming systems, or Kafka, and we've got a lot of them. We also have mobile apps that you can use to find all of our episodes by topic and to find our most popular episodes.

[SPONSOR MESSAGE]

[00:01:58] JM: As a programmer, you think an object. With MongoDB, so does your database. MongoDB is the most popular document-based database built for modern application developers and the cloud area. Millions of developers use MongoDB to power the world's most innovative products and services, from crypto currency, to online gaming, IoT and more. Try Mongo DB today with Atlas, the global cloud database service that runs on AWS, Azure and Google Cloud. Configure, deploy and connect to your database in just a few minutes. Check it out at mongodb.com/atlas. That's mongodb.com/atlas.

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

[INTERVIEW]

[00:02:54] JM: Arjun Narayan and Frank McSherry, welcome to Software Engineering Daily.

[00:02:58] AN: Thanks for having us.

[00:02:59] JM: Thank you for being here. We've done numerous shows about the subject of a data platform, and a typical scenario is a company gets started, it's got a single operational database. I like to use the example of a ridesharing company. Overtime, the ridesharing database gets a ton of data in it and eventually there needs to be an entire data platform built around all its data. Maybe there's a Kafka queue, a data lake, a data warehouse, a stream processing system, Elasticsearch. Give me your perspective on the architecture of the modern data platform.

[00:03:36] AN: I like to sort of begin by three decades ago you'd have a database and ETL tool and a data warehouse, and that actually works pretty great, this sort of separation of concerns you have for your database, for your transactional data, and your data warehouse for analytics. These two work courses have worked phenomenally well for many decades.

The challenge today is that there's a lot of use cases that don't quite fit with this traditional overnight ETL workflow, and so we've brought in other tools like Kafka queues for real-time data extraction and data movement, and then a bunch of other tools that are manually written

because the technology isn't quite powerful enough today to have that simplicity. That elegant simplicity of the database ETL data warehouse has a lot going for it.

Part of what we wish, sort of the world will look like 10 years from now is that same simplicity of having a database and ETL tool and a data warehouse comes back, because today we have about 18 different systems that are trying to get us back to that.

[00:04:41] JM: Okay. You have machine learning engineers that want to use the platform for training TensorFlow models, for example. You have infrastructure engineers who want to query it for metrics data. You've got data scientist that want to build dashboards and make reports. What are the APIs that these different users want for querying the vast volumes of data across data infrastructure?

[00:05:03] AN: I think there's sort of levels to your answer in that. Most users do want to simply use something declarative that specifies where they just specify what they want and then the system does the work. That's sort of the best case scenario. Most of the challenges that most existing tooling do a poor job of delivering what they want, and so they have to unwrap layers and start to specify how the work has to be executed.

But if you can deliver a credible experience to the end user, whoever they may be, whether they're analytics, business internal analytics, or whether they're building customer-facing experiences, and these things are getting somewhat blurry these days. Folks are building microservices. Folks are building tooling with a lot of code, because they cannot declaratively say what they want.

Data warehouses I think are the single best example of something that just does what the user ask them to the point where users often don't interact with a data warehouse directly today. The use a BI tool which spits out some declarative code that gets run on top of the data. That's the experience that I think that most other layers of the stack are trying to get to and just have not yet.

[00:06:25] JM: The data across a "data platform" is not consistent. You've got your transactional database that your users are writing to. That data is being maybe streamed into Kafka and then

it's being written into a data lake, and then you got data warehouses that are reading from it. You got ETL jobs and stuff. There is not a holistically consistent view into the data. Is that a problem?

[00:06:54] AN: Yes. That operationally ends up being a huge challenge for a variety of data users. The single best place where there is a consistent view is the transactional database, if you have a transactional database that is trying its best to maintain consistency, then you will have that.

But then this consistently often gets lost as the data makes its way through these various pipelines unless a data platform architect is being very rigorous about passing on some of these consistency information throughout the whole lifecycle of the data.

[00:07:26] JM: Are there specific kinds of applications in a company, like data applications that need to be immediately up-to-date all the time? Obviously, there's the transactional database, but are there other applications that need to be immediately up-to-date all the time?

[00:07:44] AN: Yes, and often times, today, folks are building experiences that are lagging where they're doing best efforts to recover some amount of real-time by giving up fidelity on the actual work that they're willing to do. A good example of this is, say, fraud detection. The fraud detection model for a payments company or a payments platform is often times lagging and this is something quantifiable where you can say, "Well, if we had moved the data through the platform faster, then we would have caught a larger fraction of the fraud."

This is also true in customer-facing experiences where like this seems happen to me quite a lot where you open up an app and the app is not reflecting the latest transaction you've made because you've made the transaction through a website and you just have to hit the refresh button and wait. These translated to poor customer experiences because the data is making its way through various pipelines.

[00:08:39] JM: I want to talk through some different components of a typical data platform because I do eventually want to get to Materialize and give a good picture for how you slot in

and what kinds of problems that you solve. Let's first talk about the category of distributed stream processing systems.

There are a million of these things, like we've been doing shows on them for 4-1/2 years. Okay, there's not a million, but you got Storm, Spark, Heron, Apex, Google Beam. What is the role of the distributed stream processing system and why are there so many of them?

[00:09:17] AN: Yeah. I think there're sort of two things that are being conflated by traditional stream processing ecosystem. There's the moving the data around very fast and then there's performing computations on this moving data. A lot of these technologies that you've named do in fact do some combination of both of those.

They're both interesting challenges from a technical perspective. They're both hard problems, but my view on this space is that the computations that have traditionally been possible by these stream processing frameworks has been limited. Not every stream processor you've named is able to sort of perform the full suite of computations that users would like to perform on top of moving the data around.

That, in my perspective, is why there's been this fragmentation. If you want to do some kinds of aggregations, one of these stream processors is going to perform better than another one. It ends up being a world where you kick the stream processor based upon the computations that you have decided ahead of time are mission-critical for your use case.

[00:10:26] JM: Frank, you want to give some critiques or –

[00:10:28] FM: Yeah. I don't know. I think my guess is I don't have the perspective to say why are there quite so many, but I guess is that it's just that a lot of them – This is changing, but a lot of them are relatively immature. If you go back 10 years, there are a lot of Hadoop e-clones, like people are like, "Oh! Well, Hadoop is good, but I needed to do a slightly different thing," or "I needed to go and fix that 15-second timer so I called it my own new thing. Why did Heron fork off of Storm?" There's a whole bunch of – You read the Heron paper and there's a bunch of stuff that they say like, "Oh, like Storm, we can do X." Then the Storm guys are like, "Oh, we haven't done X for two years. You just haven't been reading the most recent version of the repo."

People are just doing their own thing and it still sort of getting a read on what's really important to get done in that space.

We got other stuff like Spark, for example. Spark streaming has definitely – They had a piece of tech and they're like, "Oh, geez! We got to figure out if we can make this piece streaming." So I hit it with a stick until it looked like streaming. You got other stuff like Flink that's been around for a while that was sort of early research in the open source streaming stuff. A lot of these folks made early binding decisions that have sort of tied their hands a little bit and they've just gone as far as they can go, because unlike, let's say academia, you can't just press reset and start over on your platform. You got to go with what you have. Different people just got on different directions. If you're going to start now and say, "We want to best in breed stream processor," they'd probably look like a lot more similar and you wouldn't need quite so many different ones as we just happened to have.

[00:11:55] JM: When I think about a streaming processing system versus Hadoop, Hadoop MapReduce. I think of Hadoop MapReduce as a pretty well-defined what it's doing. It's saying, "Okay. I am going to start processing all the data in my data lake right now for this particular query and give me all the data from the data lake that fits that query."

Stream processing system, you are querying a stream of data that is "in motion", and there's more questions around when are you trying to get your answer relative to. Are you trying to get an answer starting from the time that your user actually hit the transactional database? Are you trying to get your answer from the time the data hit Kafka or the time the data hit the data lake or data warehouse or whatever? Can you explain the role of time and how time becomes kind of this almost subjective decision around which we make different data processing frameworks?

[00:13:09] FM: Yeah, sure. You're obviously right that you can think of time as this very subjective quantity in all the streaming processors. A lot of them are definitely designed to not just lock-in one particular notion of time, or like a batch processor would say like, "Right now is what when we're going to do a thing," and stream processor almost defined by this characteristic that they are able to work with multiple different versions of the world as it's changing. Instead of being in lock step with the actual step of the underlying data, keep a few different things in flight at the same time.

For sure, people have had different notions of time that they've started with and sort of move through as systems have evolved. We started with this concept that a lot of people call system time, which is just a time stamp probably, like the wall clock time or something like that, when the data actually entered into the system and that flows through the stream process. You can build a stream process that looks like that pretty easily. It just tells you on the other side, "Well, as of something o'clock, the results look like this and that."

This is great from the point of the stream processor because it's always nice internal time and it can just move things along, but people realize, "Yeah, it didn't line up brilliantly with what if someone outside the stream processor has a different take on what time looks like?" If for example your database is going through a sequence of commit IDs, you should probably – The stream processors are probably be producing results that lien up with those times or should think about the value of doing that instead of potentially a different time which is one they land in the stream processor itself. This is something that's often called event time instead of system time, which is externally imposed times.

Another example, maybe your ridesharing example, people go and click some buttons on their ridesharing app and those things periodically get set up to the company, to their data center. If someone's in a tunnel for 5 minutes, maybe after they come out of the tunnel, the data gets sent up in a big burst. That's a bit awkward for the stream processor, suddenly like, "Holy cow! I'm hearing about stuff from 5 minutes ago. This is really awkward. Maybe if I haven't been careful, I might have told people that something was the correct answer, but it's no longer the correct answer," like "Maybe I sent the police to look for Frank, because Frank fell into a pit apparently," and that's obviously the wrong conclusion and hopefully you don't build your stream processor that way. But we really do need to think about what answer we're trying to give to a person.

The tricky part from personal experience has been communicating with the user about what their expectations are. People want both correct answers. So don't show them something that's wrong, but they also want prompt answers. They don't want to stick around and wait for as long as the longest tunnel in the US to be certain that you've actually heard from everyone.

There's like a little bit of an art to that. It's not obvious that there's one thing that everyone should have been doing. For sure, I think most stream processing system started doing something and then realized about half way through that they probably should be doing something different or probably doing multiple things at once. This is one of these things that we can totally go deep for a while and I could break out multidimensional times, notions and stuff like that. But I think that's with the consent of the audience only.

[00:16:13] JM: Well, I definitely want to provide you with that consent ahead of time because I'm pretty sure that this is going to be a crucial point of discussion. But I think we should take it from a few different angles. There was a period of time where this lambda architecture was a thing, where basically the idea would be, "Look, I've got my stream processing data and I'm going to admit that this thing is going to have problems. It's not going to be entirely consistent. People are going to be in tunnels. We're going to have other kinds of lag in terms of when my data comes in. So I'm going to kind of use a combination of batch processing and stream processing. Stream processing will be for the things I need right now that may or may not be consistent. Batch processing is going to be my reconciliation mechanism that I'm going to run every three hours, two times a day, once a week, whatever."

Tell me about the issues with the lambda architecture and what we've tried to do to accommodate that.

[00:17:14] AN: Right. I think it's important to sort of look at what caused the lambda architecture. Mostly it's been that stream processors have not been capable enough. If the stream processor could compute correct answers very quickly and have them always be correct, then we wouldn't need a lambda architecture. It's being driven by the fact that stream processors have been very limited in what they could do so you have to bring in the batch processor, which is the only thing capable of reconciling the answers to the true state.

The main tradeoff here is the operational complexity is tremendous. You have typically a batch processor, a stream processor and then some third microservice that's sitting down stream of both of those that is reconciling. You've gone from one moving part to three. The operational costs are tremendous and the only folks who've been able to successfully operate lambda architectures are the really big companies with extremely large budgets and headcounts. Most

folks have often make the decision that, “Hey, we can’t do that. We’re just going to have make do with slow data pipelines, slow ETLs, waiting for ETLs to finish.”

[00:18:18] FM: A different take on that is that these are two fairly primitive tools, yeah. The stream processor you can about as a compute engine that has a 10-second timeout built into it or something like that. We’ll wait 10 seconds to get some data and then we’ll show you the results. The batch processor is on a 24-hour timeout that like every day we’ll roll up things and if we got some of our data, good. We’ll show it to you then.

Each of these are fine, and so are the only two cases you ever needed. Maybe this is fine. But in a lot of situations, things are a bit more fluid than that. The data, for some reason – And maybe once every hour there’s a little bit of a hiccup at your upstream backup and it takes a little while to get you the data. Generally, it’s actually current to within a second.

Having to have that one hour actually be something that fails over and you only see it after the 24-hour drop back to the batch processing thing because it exceeded some of the built-in latencies or the stream processors, suboptimal. Absolutely, people can take these architectures and turn some knobs and change them around. There’s a benefit to having a slightly more fluid system that can handle these sorts of hiccups as they come and go and give you the tightest experience that it can manage. If that means that there’s some data that comes in late on an hour by hour basis, handling that gracefully is an important part of real grownup mature system.

[SPONSOR MESSAGE]

[00:19:43] JM: Today’s episode of Software Engineering Daily is sponsored by Datadog, a monitoring platform for cloud scale infrastructure and applications. Datadog provides dashboarding, alerting, application performance monitoring and log management in one tightly integrated platform so you can get end-to-end visibility quickly and it integrates seamlessly with AWS so you can start monitoring EC2, RDS, ECS and all of your other AWS services in minutes.

Visualize key metrics, set alerts to identify anomalies and collaborate with your team to troubleshoot and fix issues fast. Try it yourself by starting a free 14-day trial today. Listeners of

this podcast will also receive a free Datadog t-shirt. Go to softwareengineeringdaily.com/datadog to get that t-shirt. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:20:45] JM: One other moment in time before we start getting into what you're doing with Materialize, the DataFlow paper that came out of Google – My understanding of this paper is that it's formalized the role of time as an abstraction in terms of event time and processing time and these things like windowing and watermarks and basically these abstractions that give you a framework for reasoning about time. The DataFlow paper basically acknowledged the fact that we cannot have this perfect notion of consistency at all times from a streaming system. Instead we're going to use these time abstractions to build a system of reasoning.

You can correct me if I'm wrong and just explain what the role of the DataFlow paper was and perhaps why it didn't solve everything in stream processing.

[00:21:44] FM: I definitely have – I have a take on that, which you should ask other people for their take as well. The work that I've done sort of around the same time was on this Naiad system, where we used more advanced notions of time and show up in the DataFlow paper. In particular, times are allowed to vary in multiple dimensions and superficially you can say like you could have a time that has two coordinates. One is system time and one is event time and you can talk about progress through a stream in either dimension at once. You don't have to move forward either in system time or an event time.

This is the thing that's basically tied up just about every other stream process out there is they have like, "Time is going to be an int." Now that time is an int, oh, should it be system time or should it be even time?" This is really complicated, and it is complicated because if you pick either one, there are various scenarios that cause you trouble.

If you pick both at the same time using these more advanced notions of time, then the assertion that you have to sacrifice either consistency or responsiveness is not right. You can get both at the same time as long as you end up with a system that very responsibly tells you everything it knows about the state of the world, which happens to not always be current up to right now. You

can decide as the user what you want to do with that. You can either say, “Oh, I’m going to have to wait till I see the right answer because I’m part of the auditing firm and we need to know the right answer,” or you can say we have about a second and a half to show user and advertisement. Are they going to leave? Let’s do our best effort.

This can totally be up to the user of the stream processor rather than a built-in aspect of the stream processor itself is my take on that. Various other things, like windowing. Windowing is cute. It’s not mandatory for stream processors. You can have stream processors that don’t use windowing. It’s very helpful.

If I recall correctly, the DataFlow paper and a bunch of other stream processing systems are append only. So you can’t talk about retracting elements from a stream. If that’s the case, you kind of got to put windowing or something like it in there to throw away data. But other systems support retraction. So stuff like that, and it’s not mandatory anymore.

[00:23:45] JM: As we get into talking about Materialize, I want to define your namesake. Could you explain what the term materialized view means?

[00:23:56] FM: A materialized view in database land. Okay. Often, SQL database, say, lambda. A view is a description of a query. It’s sort of a named quantity that presents essentially as it if were a table or something like that, but it’s not actually a table. It’s not a collection of records that a person has put in to your database. It’s a way of determining a proxy for a table. Though in a traditional database, a view is literally just a description of the query. It’s not actually any data.

In something like SQL, you can ask for a materialized view, which essentially instructs the database to go and determine the results, like figure out what should the actual contents of this would be if you’re going to evaluate it right now, and thus materializing the results and making it pretty quick if you want to go back and look at the answers.

Traditionally in a database, if you asked for a materialized view, it computes the results and just sort of leave them there because, “Oh, it’s expensive to update them if the input data change.”

You can ask for that. You can ask for like on-commit refreshes and it will go and recomputed your view for you, but it's expensive in a lot of traditionally databases.

Sort of the goal with Materialize, the company and the thing that we're building, is to give people this experience, but with a continually up-to-date, like always fresh view on all of their data, all of their views.

[00:25:08] JM: In a modern "data platform" – Actually, we just did a show today that pertains to this about there's a challenger bank called Nubank, and their data platform, they have all this work that needs to go into preparing data from OLTP data and third-party data sources and all these other things and then they periodically use Spark to create these materialized views, and these materialized views, I think they're called data frames in Spark world and they're eventually delivered to the end user who is an analyst or a data scientist who can finally query them with SQL.

I'll say that just to kind of tee you up for talking about Materialize, because my understanding is that Materialize makes that process of getting a data platform into SQL quriable formats and consistency much easier. Tell me what Materialize is.

[00:26:13] AN: Materialize, like what you just described there, is a way to get all of those nice things that we've been talking about, like those materialized views without having to write a lot of code that specifies how that result is to be computed. A lot of the moving parts that you described today, which is get the data out, write these Spark jobs, have them be maybe go in and manually rewrite some of these queries so that they're more efficiently incrementally computable. All of these you could achieve today if you trade latency by ETL-ing it into your data warehouse and writing a nice SQL query. The SQL query doesn't say anything about how to compute the result. It just says what is the answer that I want? You write that very declaratively. We want to maintain that user experience.

With Materialize today, you can write down that same SQL query and then it's Materialize's job to figure out what underlying DataFlows have to be built. What has to be incrementally maintained so that the user can still live at this nice clean abstraction layer of thinking about their business problem, like what exactly is it that I want? They may not even have to write SQL

because they may be living in a BI tool where they're pointing and clicking and creating a dashboard that is very important to their business, which under the hood is being generated into this big, gnarly SQL as auto-generated SQL often is.

You remove this layer of operational complexity where you need to hire expensive data platform engineers. You need to sit down and write down all these pipelines and thousands of lives of Java or Python because you can just live up in the SQL world.

[00:27:50] JM: Do you want anything, Frank?

[00:27:51] FM: No. I mean, Nubank is definitely a great example. We've chatted with them going back – Folks that have been excited about the tech underneath differential DataFlow and stuff like that. They have some 6,000 queries that, yeah, they basically need to them incrementally – They like to have them incrementally maintained because it's just an operational pain in the butt to come in every 24 hours and see did the job complete or didn't? These things fail for reasons you don't understand. Then maybe around 48 hours still data and just – It's unpleasant and kind of distasteful to have to run your operations that way.

If all of your people, your analysts, are skilled people and thinking about their problems but they'd like to express them various ways. I think Nubank is using like Clojure and sort of Datomic style ATMs, but if you have similar – This is the next high-level with specifying your problems sort of like SQL in the sense that you don't really want to have to be sitting there and moving around data yourself.

Having access to data engineers is great, but if there's that latency on your path from thinking about like, "Oh! I could really use the answer to this particular question. Do I have to throw it over the wall and have data engineer write it up for me? Maybe tomorrow I will start to see the answers versus type it in and maybe in a few seconds you see the answers, and then I'll start to spill out real-time after that." It's just a big qualitative leap there in terms of what you can allow your analysts to do.

[00:29:14] JM: Now, if I'm familiar with the typical process of a data platform and I know that there's like 15 different systems between the OLTP transactional data that is being written to like

the ridesharing database or whatever, like the rides that are proceeding, how much they cost, etc., and the data scientist that's actually making the SQL query. What's interesting about what you're trying to do is it sounds like you're compressing that volume with different systems into something that's easier to work with. Can you contrast the architecture of Materialize with the typical data platform, like something like Nubank?

[00:30:00] FM: I won't pretend to know all the ins and outs of Nubank's data architectures and I don't want to oversimplify the complexities that actually exist out there. The goal with Materialize is absolutely not to delete everyone's data platform and replace it with – I mean, that would be wonderful, but there's enough weird stuff going on out there that the reality is – What we love to do is slurp up – I don't know, let's say 95% of the – Onboarding little microservices that people are writing because they've just realized like, "Oh! Geez! I really need to maintain in some key value store the current accumulation of some quantity. Let me just zip that off and post it as a microservice." It's a lot of things like that that you could stitch up all in one coherent framework, Materialize, let's say, where you could write in the order of tens of lines of SQL to describe what you want instead of a whole bunch of different jars that you ship all over the place.

Grabbing this, like thinking of all the little PaaS that your data might flow through your data platform, finding some nice connected component that's relatively sizeable and is contracting that down to one platform. Instead of data burbling through 15 different steps, if all of these steps can just be consolidated into some SQL roles, this would be great. Then any people who are consuming these data products in the organization can be sure they're getting consistent views. They can receive prompt information about how current are they instead of crossing their fingers and like, "Oh, it's 15 seconds for each of these things, and this thing ticks over every hour." I think you get a much more sane integrated experience.

Okay. You're going to have to like – We're not going to let you – TensorFlow doesn't run in Materialize yet or anything like that. If you want to do machine learning type things on top of that for the moment, you're absolutely going to need to extract your data and work with it there. But the hope is that for a large fraction of business analysts needs the SQL sort of class of queries that there'll be nice integrated story there that other people can tap into with their other exotic use cases.

[00:31:52] AN: And sort of not to oversimplify things, I mean there's a lot of reasons why they're separate systems in the data platform today. I mean, there are a lot of use cases where people actually don't necessarily need real-time updates if it's quarterly reporting. It's okay for that to be done off of us. In fact, what you might care about that is making sure you wait to get a consistent snapshot of the data because it's okay to deliver their report. If you take a few days to deliver that report, it just has to be correct.

For a machine learning engineer, a typical workflow might just be downloading a static copy and iterating because you don't want that every time you recomputed the answer, because you may be debugging your model. You don't want the data actually changing when you are iterating on your model.

There're also good reasons for the OLTP database to be separated. In fact, I think the OLTP database is the thing that's probably going to be kept separate forever. There's a nice joke that your OLTP database, you'll have somebody watching it and a dog to watch that person to make sure that they don't touch the OLTP database.

Very important resource separation because you can't just have an analyst sort of waltz in, add a few indexes, slow down all the writes and all your transactions now take 5 times longer than they used to. So you often do want to maintain the separation of systems where, step one is the OLTP database stocks are very restricted set of customer-facing, low-latency critical user-facing things. Then stuff is ETL'd out or this change data capture that's coming out of that OLTD database. Then now we're in the zone where all the other stakeholders of the company can have free access to this changing data. But it's important to realize even in that world, you are going to have separate systems here in order to maintain separation of concerns.

[00:33:32] JM: If we're talking about getting the data from the transactional database into a place where the data scientists and data analysts can query it, I'd like to understand the architecture for what you're reading. Are you reading the database write-ahead log? Are you taking a snapshot of my transactional database? How are you getting data into your system and what is the series of steps to getting it there?

[00:34:02] FM: A lot of transaction processors, in fact I'd say most of the series ones have the ability to spin up read replicas, for example. You have the main transaction processor that's actually handling. It's the source of truth and is able to feed what's called a change data capture out of the database to inform, let's say, read replicas, places where people can go and get read access to the data without impacting the main transaction processor, and they have machinery for this. MySQL has a bin log that it sort of produces. It talks about what are the committed transactions that have gone on in the database. It's used both for recovery and also just other people are going to look at it and get the record of what the database agrees has happened.

At the moment, we're using some other tech that people have done. There's a thing called Debezium out there which interfaces with a bunch of different databases and tries to put the results of these logs into some common format that it drops into Kafka. Kafka is one of the places we can read data in a somewhat unified form now and we sort of step on to that Kafka BUS, if you will. Materialize has a freestanding decoupled system that is pulling data as quickly as it can out of Kafka usually limited by the speed at which it's put in there and looks now like a replica of the database.

A little bit of like, because it's not literally the exact same system as the transaction processor, but the appropriate use of logical time stamps and stuff like that to keep us informed about like what are we really surfacing. What transactions are we revealing? It keeps everyone sane at least.

[00:35:38] JM: Okay. Tell me more about this. Let's say I get all the data from my transactional database into Kafka and then it's eventually being read into – Is Materialize a database? Is it an in-memory system? What –

[00:35:56] FM: Yeah, it's a good – This is totally the same question. In a lot of ways we look a little bit like a read replica from the outside in the sense that you can show up and you can start asking us questions about like, "Hey, here's an analytic query I'd like you to answer for me. I'd like to get some views. I'd like to get some indexes," but you're not allowed to do inserts or deletes, it's because we're mirroring this other source of truth.

But internally we look very different from a traditional database. We look a lot more like a data parallel DataFlow processor at that point. We have large number of worker threads spun up potentially across multiple processes if you've got that turned on. Internally what we're doing is we're taking each of your queries and instead of processing the way a database would process it, which is this pull model, where you sort of go through and you start trying to work your way through the query, surfacing records as you go.

A lot of the DataFlow settings that the data parallel compute type stuff build DataFlow graphs, which say – Essentially, it's like a push model which says, "Starting from our inputs, imagine we're going to force all of our data into these DataFlow graphs. We're going to join these two things together and do reduction and join it with a third thing. Build out a DataFlow graph that describes them and start pushing what are essentially these changes into this DataFlow graph." A lot of the changes at the beginning are just saying like, "Oh, previously, we didn't have any data. Now we have some data."

But once that reaches sort of a steady state, the changes start to look like additions and are tractions of data. If someone's updated a tuple in the transaction process or we're going to get something that's says, "Oh geez! This value has changed. We had previously have been into a max. We're going to have to figure out how to find the new correct answer. Be darn sure that we get the right answer out."

A fair deal of cares have been put in to making these DataFlow graphs be responsive. If a single record comes in changing one of the inputs, we want to make sure that we do proportionately small amount of work propagating that through the DataFlow so that you get the refreshed answer as fast as you can.

Internally, this looks a lot like a more big data E data parallel processor with lots of sort of sharded workers and sharded DataFlow operators that exchange data between them and a little less like the way an analytic processor look.

[00:38:04] JM: If I think about my transactional database, it's got like tons and tons of users. It's got tons and tons of – It's got records for each of those users. It's got – I don't know. If it's a ridesharing database, it's got rides that are going on. How do I figure out what data – Or I

should say, how do you figure out what data is materialized in that DataFlow graph? Because if I was to keep the entire database, the entirety of my database in that materialized graph, that's probably going to be really expensive, right?

[00:38:42] FM: I mean, it could be. We give you the ability – When you turn Materialize, and in fact when you turn it on, it's not bound to any particular source of truth database. We have you create sources essentially in Materialize which announce, "I'd like to go and I'd like to grab the stream of changes that correspond to some table and some database or potentially a set of tables and a database," and you definitely don't have to mirror the entire database. Though what we are going to do almost certainly is if you identify tables that you want to pull in, we're going to end up almost certainly mirroring what's going on in those tables.

If it turns out that you want to have filter project before capturing that, so if you want to pull down your customer's file but you really only need – For whatever you're doing, you really only need four of the columns, let's say, because you're just trying to figure out their location and some billing information and you don't need their last 27 logins or something like that. You can project out all these columns before they get materialized. But, yeah, no joke. When you pull down the data, it's for these fast incremental updates, we're going to be wanting indexed in-memory representations of all these data. If you got a join going and one record is up, we want to very quickly figure out, "Geez! What is that hip? What are the implications of this new record and one of the inputs to the join." We don't want to go back to the transaction process or/and burden it with a question of like, "Could you look this up for me?" or anything like that. We're going to end up mirroring those parts of the DataFlow graph that are crucial for reacting promptly to changes.

[00:40:05] JM: Let's take this from the user's perspective. I am a data analyst. Let's say I want to create a materialized view and then query it. Take me through – Maybe you can invent a query for this ridesharing system and take me through how the materialized view is created and the life of a query.

[00:40:26] AN: Right. A user of Materialize today – Materialize looks, feels like PostgreS. They can use a PSQL, the out of the box PostgreS client and then they connect to Materialize. The first thing Materialize wants to know is what data are we talking about? So you can create these source, like Frank mentioned, which is akin to a create table statement. It says, "Go connect to

that Kafka topic where you will find some Avro or some protobuf or some JSON and get all of that. Here's a scheme," and you can use the schema registry to have that auto-populated or you can manually specify schema or you can – PostgreS has lovely support for JSON, and we follow that dialect. You can say, "Oh! You're going to find – Get everything and call that one single column with a JSON blob in it."

Then you can progressively unpack it by creating some views over that. So you could say, "Well, now that you've got this one source that is JSON. Creative a view over that where you should find a column named name, and I want you to get that out and interpret that as a string." You might a column named account number. Interpret that as an int. It's very much like you write a select query where you're selecting as that table and you play around with that a bit and you say, "Ah! That's a really nice query. I want to keep that around," so you could copy up that select query that you like and prevent it with create view as select so and so.

What's nice is you can chain these views in the same way as you may – In the past, you've had to chain microservices where they're progressively unpacking data. The other thing you could do is you could take five of these sources and join them together, because you may have a – They maybe actually be coming from five different tables. There may be a user's table. They may be a transactions table. There may be a driver's table. You may want to create a view which is what are the last 10 – I don't know. Which is the sum of the costs of the prices of the last 10 trips per user? That may require going by the user's table, the payment's table, the ride's table, however you've organized your data, and then joining across all of these things.

Traditionally, stream processors in the past have not been quite capable of that. The join is the hard part. In Materialize, you just write the SQL. That's A join, B join, C on your join condition. Of course, Materialize is going to have to keep some state around in order to surface those joins. Part of the – If I'd to identity one instead of three or four core innovations is sort of being able to do that without really breaking a sweat even if the state to be managed is quite large.

But from the user's perspective, I mean, they don't really care. They don't want to care in the unfortunate position of today's tech stack of having to care about these questions, but with Materialize, they just write the SQL query. Then eventually the final view that they have, they could say, "Hey, Materialize this view eagerly." Instead of create view, they write create

materialized view. Then they have one or two options, which is they could take all the changes and have Materialize eagerly push them out to some streaming message BUS. They could have that be sent right back to a new Kafka topic or they could connect to it and run select queries against that view where the user just says, “Select star from whatever that materialized view name is using, again, a standard PostgreS connector which is presented in pretty much any framework or language.

What’s nice is from the user’s perceptive, from the app implementer’s perspective, they’re just writing SQL queries in the PostgreS dialect that they’ve probably been familiar with from over a decade and they’re declaratively specifying what they want without having to say, “Well, this join is really hard and you got to manage the state and have to build a data structure to manage all the state and I’m going to build a microservice to do that and I’m going to put that in some NoSQL store.” That’s sort of traditionally been the answer to build these large stateful data pipelines. Instead they say, “Nope. Here’s the SQL I want. I just want this to always be live.” They interact with it very much like they would interact with a data warehouse, except that it’s real-time.

[00:44:28] FM: I mean, definitely one way to think about it or like the ideal look and feel from my point of view is this is like a database accelerator in some sense. You use exactly the same way you use it for BI in the past. It’s just it goes really fast for some reason. That reason is essentially that you’ve engaged in this contract with us that you’re going to be interested in the same thing overtime. Like a standard analytic platform is surprisingly good at taking queries that had no relationd to anything you’ve asked previously and grinding through them really quickly. The value proposition here is if you want to look at something that’s pretty similar to what you looked at before, like the same view, we’re going to be surprisingly fast at giving that live for you.

What we found is that there are a lot of people who – Yeah, that’s what they want. They’re using dashboards. The dashboards are asking the same questions over and over again or they’re tracking various metrics. They’re doing monitoring of other sorts rather than hit – They want the same experience, but rather they’re hitting an analytic process that has to go and work quite hard to figure out the correct answer. Again, just getting a little thing every time there’re some

changes or otherwise a concise representation of what the answer is without doing all that work over and over again. But same look and feel really is the goal.

[SPONSOR MESSAGE]

[00:45:46] : DigitalOcean makes infrastructure simple. I continue to use DigitalOcean because of the low friction and attention to user experience. DigitalOcean has kept the experience simple and I can spin up a server in less than a minute and get high quality performance for a low price. For an application that needs to scale, DigitalOcean has CPU optimized droplets, memory optimized droplets, managed databases, managed Kubernetes and many more products. DigitalOcean has the flexibility to choose the right instance for the right workload and he could mix-and-match different configurations of CPU and RAM.

If you get stuck, DigitalOcean has thousands of high-quality tutorials, responsive Q&A forums and a customer team who treats customers respectfully. DigitalOcean lets developers focus on what they are building. Visit do.co/sedaily and receive $100 in credit over 60 days. That $100 can be put towards hosting or infrastructure and that includes managed databases, a managed Kubernetes service and more.

If you want to get started with Kubernetes, DigitalOcean is a great place to go. You can use your $100 to start building your distributed system and you can get that $100 in credit for free at do.co/sedaily.

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

[INTERVIEW CONTINUED]

[00:47:22] JM: To revisit what we discussed earlier and take a step away from the Materialize architecture for a second, the problems inherent in the stream processing systems where you have like a rider who goes into a tunnel for a bit and they lose their connectivity and you don't hear from them for five minutes, and therefore whatever SQL query is querying the data that has hit the data platform, that SQL query is basically invalid for those five minutes. Then the person comes out of the tunnel. They reconnect. There's a big burst of data that hits your data

platform and then suddenly your – Whatever standing query there is suddenly gets updated and reconciled. That issue still is going to exist with Materialize, right?

[00:48:14] MF: Yeah, absolutely. Yeah. There's no magic that figures out – Like those five minutes, the world has to wait to figure out what the right answer is there. The best you can hope for, for our take at least, is the best you can hope for is clear answers about what you are seeing. For example, getting interactive – Working out the UX for this particular dimension is a bit wonky, but like you ask a question. Getting back information, it says, "As of right now, system time. This is the spectrum of correct answers. We can tell you for sure about things from five minutes ago. We can't tell you for sure about anything before that, but if you want to stick around for a little while, we can get you that answer."

What an answer actually means when you get it back, what is it current with respect to various time dimensions is something that fortunately we have the tools to reflect back up to the user. Figuring out how to inform people what they're actually seeing is a little crazier, because different people have different needs. You want to see the car on the map moving even if you're not really sure where it is. So make a guess is one requirement for an app. How much did you charge my credit card at the end of the trip? That's a very different thing and you kind of want to wait to see how that actually worked out before you show it to someone.

We do want the underlying technology to be able to handle both of these cases. The UX side of – If you just type in select my money from account and press enter, how long should we wait? What you actually want to see is little more awkward. No particular magic yet thinking through what – How we want to surface this up to people. But internally, we know exactly the correct answer at various different time dimensions.

[00:49:52] JM: The applications that you have built with Materialize, I understand that there's some connection to the Naiad paper. You've mentioned this once. There's also this timely DataFlow project. Can you explain the circumstances that led to the creation of Materialize, the "computer science" that led to this thing?

[00:50:15] FM: Sure. There's some technology that led to it and then there's some social things that actually resulted in the formation of the company. Technology, way back when. So many,

many years ago at Microsoft's research lab here in Silicon Valley. I supposed we're in San Francisco now, but here relative to Microsoft up in Redmond. There are a really cool group of people that are working on a lot of really interesting problems. One of them, for example, this is where DryadLINQ. I don't know if anyone's ever familiar with that who listens to this, but this is like a precursor to Spark. It was like four years before Spark came out, there was already in C#. That's LINQ, like language, which is essentially these idioms that you have datasets with dot select, dot filter, dot – All these sorts of things on them, dot join. And you build up DataFlows by tricking people into expressing things declaratively. That team did a lot of really great stuff and are mostly at Google now.

But, while there, took a rev on that, which was this Naiad project that was, "Oh! Surely we could be able to do similar sorts of things, but with –" Initially, iterative computations is actually what we're looking at. It was like putting loops into computations. But that pretty quickly turned into stream processing. The interesting thing about iterative computation is that you have some state that you can return to relatively quickly. Spark-style computations can handle of the state, but each time you come back to that state, you're going to load it back up again and it's kind of slow.

Anyhow, at Microsoft, there's this Naiad project that was really the genesis of this timely DataFlow work where we sort of went pretty deep on what if we commit our computation to a really strict DataFlow graph? Pretty expressive one, but like serious business. We're going to hold on to this DataFlow as our guiding principle. When you think about that for a while, what you end up with – What we ended up with at least was this timely DataFlow model where data circulate through DataFlow graph center all time stamped with logical time stamps. That's fundamentally new. But the system underneath it took advantage of this in ways that people hadn't previously built off [inaudible 00:52:12]. It had some really nice performance properties and went fast and then within the year, Microsoft dissolved that lab and we all got scattered to the wind.

I wandered away and took a few years of vacation, sort of surfing in Morocco for a bit and programming and Berlin and just doing random stuff like that and picked up RUST and started using RUST, which is great. I like RUST a lot. Other people might not, but it really worked for me. I was happily typing up timely DataFlow, differential DataFlow stuff and having a fairly

simple life, which when Arjun showed up. Arjun – I don't know. I probably shouldn't tell his story for him, but had been following this work for a while. To my understanding, found it very stimulating and interesting and liked it and he can embellish that if he likes, but came up with the observation that this is great. You typing on this system is adorable, but if you actually want to see if it has any legs, there's going to need to be people who do annoying bits of work. Get people who write interop players, people who write documentation, people who do testing. You're not going to do that stuff. I mean, Frank. The right mechanism to cost this app and it's actually to form a company. Put together something that is going to attempt to turn the good things in this space into value for people and form the cycle where you then get to pay people to make sure that this is actually a good product that is actually as valuable as it can be to people in the enterprise infrastructure, for example. A positive virtual cycle happens where you make more people happier and happier and you make the world, at least of enterprise infrastructure, a better place.

[00:53:45] JM: This is a very computer science explanation of what a business if.

[00:53:50] FM: I tried to make it sound as good as possible.

[00:53:53] JM: No. It's beautiful. Actually, very objectively and accurately defined.

[00:53:57] FM: I'm going to pause there and let Arjun correct that as he sees fit.

[00:54:01] AN: That's a lovely, adorable story. Here's the real story. That was a lonely grad student working in the research minds, actually on differential privacy, which Frank co-invented before starting Naiad, the Naiad project. When I come across in the research field that I was on, which is distributed systems, the sort of seminal works that were happening while I was in grad school were the Spark paper, and then the Spanner paper, and then the Naiad paper, which sort of one of the big award-winning papers around 2011, 12, 13 while I was a grad student.

As I was following all of that, it seemed a lot more interesting to me than differential privacy, and that's speaking purely personally. I had been following the Naiad work since then. I eventually went on to work at Cockroach Labs, because I found both projects on sort of very different areas of sort of the data lifecycle.

[00:54:59] JM: Cockroach Labs, being Spanner.

[00:55:00] AN: Cockroach Labs being directly inspired by Spanner. CockroachDB being directly inspired by Spanner. But I always, always thought that CockroachDB is great, and if it succeeds, it will have solved the OLTP side of the data platform equation. There're all still all these other stuff there that really is looking to be cleaned up quite a bit, because from my perspective it looked like a more as of microservices that's extremely painful.

From a software engineering point of view, what Frank had essentially built over about four years in RUST was akin to a – The analogy I would use is he had built the query execution engine and nothing more. A query execution engine in a database is not sufficient to be used as a database as all the other parts of it, as the language, the parsing, the planning, all of these things that still remains to be done. From my perspective, that could only really be fully achieved with commercial backing, because the scope of the engineering required to get that done is beyond a single person's capabilities.

[00:56:03] FM: Or interests as it turned out. It was like as a person who's just happily occasionally going suffering and occasionally having beer in Berlin or whatever, like the idea of working really hard to have a standards compliant parsing. Actual SQL 92 and all of it warts doesn't make you happy. But when you actually think about we're going to make it a product, then there's now like a proper motivation to actually say like, "This is valuable if we can do all of these sort of gross things and remove pain that other people are actually feeling right now."

[00:56:38] AN: Right. One of the sort of product decisions that we've made is that Materialize looks like PostgreS to the end user and that's a very much a compatibility decision, because if we truly want people to get access to a real-time data, they are going to do so from – They may not be doing so directly. They're going to do so from BI tools, for instance. BI tools know how to speak to PostgreS. We don't really want to get into this business where you've invented a new dialect of how to talk to your stream processing framework and have to go and rebuild all these integrations layers. In fact, you just want to have dropping compatibility and just straight up lie to the system that, "Hey, as far as they're concerned, they are talking to PostgreS. They are talking

directly to the OLTP PostgreS database. They don't ever need to know that, in fact, no DBA in their right mind will allow the BI tool to hammer the OLTP system."

But nonetheless, that's the easiest way for the BI tool to live. That's the sort of architectural decision that requires some amount of commercial backing, because truth be told, PostgreS has a lot of rough edges in terms of parsing, planning, translating those queries down to the actual DataFlow graphs which you really need to get to 100% compatibility because otherwise the BI tool is just going to ever route.

[00:58:01] JM: Tell me more about the process of taking a research paper and productizing it.

[00:58:09] AN: One nice thing about academia is there is a lot of similarities to the entrepreneurial journey of spending a lot of time prototyping, testing, iterating. What I would say is it's not always clearly a good thing to build a company around software. There has to be market need, and I think there's often times research projects that area shoehorned into a commercial venture without there being an obvious need for that product to exist.

I think the most important thing is to step back and sort of look at it from does that thing actually need to exist? Will people pay for it? Do they actually care? Questions like that. I think some of that was sort of inherently achieved by the fact that I had nothing to do with the Naiad project. I was just looking around for – I was feeling the pain as a software developer for what big data engineering things needed to be fixed. From my point of view, it was OLTP and streaming sort of real-time OLAP.

Spanner had this commercial backing of Google and they had sort of internally faced some of those pains of NoSQL which drove the development of Spanner. But I think one thing that was helpful was I had no particular affection for Naiad out of having sort of built it myself. That was all Frank. But I had come at it from, "The market clearly needs this. I need this because I am trying to build applications that are effectively materializing various views and it's incredibly frustrating to do manually via gluing a bunch of microservices together," and sort of went and dragged Frank and yelled at him a bit to maybe take this a little bit more seriously than a random open source project.

[00:59:55] FM: The next thing that I will say, academia – There's a spectrum and on one end you have the purest of academic research [inaudible 01:00:03] stuff. On the other end, you maybe have industry and there's been a lot of smearing in between with a lot of open source projects where like a big transition for me personally after leaving Microsoft was putting stuff out there in the open, open source stuff and doing open development out there.

I think people come back and complain when it was like there's some [inaudible 01:00:21] corner that was supposed to work but didn't that in academia you're just like, "Oh, yeah. It's fine." Either we can fix that or who cares. It's not even a big deal. Look at our plots. Actually trying to fix these, these sorts of things. Personally, my part of the journey like put me on a bit more of a path of like, "Oh! Some of these details actually matter. They're not going to get you a fancy price or anything like that." If your system has a bunch of a few clever ideas and nothing else holding it together, it's not nearly as sort of rewarding, I guess, as something that actually does what it says in as many cases as you can manage.

Personally, it's heading in that direction already. Like the whole business need and stuff like that, that's beyond [inaudible 01:01:02] still at the moment. But getting a lot closer to like actually solving real people's problems, it feels good. Making academics happy is not as satisfying as you might think.

[01:01:13] JM: We've glossed over some of the architectural details, because you can't really discuss them in an hour, much less on a podcast. What have been some of the acute challenges of putting this thing into production of building a production system that serves queries the way that you want it to that has the correct user APIs that you want them to? Tell me about just the process of productionizing this system and just the nitty-gritty engineering pains.

[01:01:43] NA: Right. Some of these, with a little bit of forethought can be made less painful. For instance, we're working with a series of design partners who are giving us the sort of feedback, early feedback before we sort of put this out there for anybody to use. The second this is an intentional decision to very carefully control the deployment environment.

Materialize runs fine on your laptop. That's not the intended production use case. Building an I eye towards owning and operating a cloud service from day one as being the only serious

production environment that we consider, it really helps narrow down the list of things that could possibly go wrong. You don't have to worry about what will people's experiences be running on some weird operating system with some weird – We carefully vet the production-ready environments that Materialize can work on. Then hopefully the users experiences, they don't have to care about any of these things. They talk over a network port to SQL just as they might to, say, Aurora, or Cockroach.

[01:02:51] MF: On the like nitty-gritty side, there's like a few things that come up that are – I don't want to say unanticipated, but they're potentially surprising if you come at this just from I'm going to build a database point of view. If you try to turn standard database queries into streaming DataFlows, you immediately have a few challenging, exciting problems. One of them is you've got to make each of your queries into a DataFlow, and you can write some weird stuff in SQL. You can write all these deeply correlated sub-queries that have to go through a deep correlation process if you actually want to turn them into a nice flat relational plan.

In some systems, there's give up and there's like, "Oh, it's too complicated. Essentially, let's just shell out to the sub-queries we can actually execute just as queries against the database itself. It's fine," if things are too complicated, and we just don't have that choice. We have to actually figure out how to get 100% conversion for SQL queries into DataFlow. Otherwise we're just not viable on that class of queries.

This involved some great work by some of the engineers that we have. We've been really helped by working with some pretty great employees, the engineers that we have have come in with a lot of really solid background on, "This is this part of the database [inaudible 01:03:59] look and feel," or we've got some other folks who sort of deeply understand relational algebra and deep correlation and stuff like that. This has really smoothed out a lot of the bumps that would otherwise potentially surprise people if they just started to try to hit these together with hammers.

Yeah, there're other fun questions like query optimization for a database is a little different than for a stream processor. In a database, you're trying to get the answer back, like analytic processor. You're trying to get the answer back as fast as possible. We'd like to get you the answer back as fast as possible too, but we're definitely very concerned about the standing

memory footprint of a query. Because if we're going to maintain this, we need to keep some stuff resident to memory, and we could plan a query maybe a few different ways and if one of them has a terabyte of data just sitting there hot that were constantly doing random accessing to and a different one just has a few gigabytes. The few gigabytes one is pretty appealing.

For example, one of the nice benefits of using timely DataFlow as supposed to some other stream processors is it has some functionality that will allow you to share state between different DataFlows, and this is really helpful. So if you have your customers relation and your customers have a customer ID, which is their primary key, we're going to go and index that stream of changes by that primary key and anyone who needs to use that customers relation, any DataFlow that needs to use it can share the same in-memory asset across all these DataFlows. If you got 20 analysts who are trying to spin up queries that involve let's just say standard relational primary key, foreign key type things, it would be too bad if each of those 20 people had to create a new DataFlow with independent memory requirements that each mirrored the sources of data that they're working with. That's what a lot of the stream processors would do at the moment, which is one of the reasons probably that they haven't taken over as like a scalable, scalable in the sense of number of analysts infrastructure. Whereas the hope here is absolutely you can point to hundred analysts at Materialize, and although they might have different questions that they're asking, if they're using the data in the same way, essentially if they're joining on foreign keys of – If you have like a ride descriptor table that has a customer ID and a driver ID in it joined against customer and driver tables, we're not going to need to spin up any new in-memory assets for that.

This is like one of the things, it was sort of – Not so much a speed bump, but like, hey, fortunate consequence of using by the time they did it for the tech. But it's the sort of thing that would definitely be a bit of a hiccup if you tried to turn this on in something like, let's say, Flink or Storm.

[01:06:23] JM: Let's close off with a little bit of discussion from the go-to-market point of view. Yeah, Frank, the chief scientist I believe is pointing to Arjun.

[01:06:35] FM: Definitely not chief go-to-market expert.

[01:06:38] JM: I've talked to a lot of companies in various areas of the data platform world and I definitely know that like the buyers are ready to say take my money in many cases, but you obviously have to provide them some value. You have to go through this process of kind of proving things out. Tell me about the go-to-market strategy and how you convince people to use Materialize.

[01:07:04] AN: Right. One thing we've been somewhat surprised by is just how much appetite there is for real-time data processing infrastructure that actually works. We've spent the last year quietly working on building Materialize without much of a public presence. But a large part of our intention is to make a version of Materialize freely available to developers to prototype on their laptops. A lot of what we've been surprised by are sort of the new use cases that come up. This isn't [inaudible 01:07:36], but in the cloud, right? There's a set of well-defined pieces of infrastructure that a well-scoped people know what they expectations are.

One of the nice things about something like Materialize is it enables things that haven't been possible before. It's more akin to something that developers really want to get their hands on and play with. Materialize has a free and source available tier, not yet, but perhaps by the time this podcast is published. Our belief strongly is that folks are going to want – This is a serious piece of their core infrastructure. They're going to want help with productionizing it.

Commercially, Materialize is available only as a hosted cloud service where if this is part of your production infrastructure, you're going to care about downtime. You're going to care about high-availability. These are things that we will gladly take off of your hands, because we're the experts at running and operating Materialize.

[01:08:35] JM: Maybe to close off, the strategy of open source versus not open source. I think it sounds like you've gone more with the Snowflake direction of close source, kind of this is a complicated piece of data infrastructure. We're going to abstract it away from you and give you a managed service, but there's not going to be an open source ecosystem around that?

[01:08:56] AN: There is – I don't know when this podcast is published, but maybe there will be already a source available version of Materialize.

[01:09:03] JM: Oh, okay! Awesome. All right. Well, that's great to hear. Guys, it's been great talking. Any closing thoughts or –

[01:09:10] FM: No. This has been super fun. I mean, you've drawn out a bunch of really cool things that's great to be able to talk about. Plan is in the near future to be able to tell people even more going forward and people should be looking for that. Basically as soon as this goes live, we'll start putting out content.

[01:09:24] JM: Awesome. Well, I will be following you guys closely.

[01:09:27] AN: Thank you very much for taking the time.

[END INTERVIEW]

[01:09:37] JM: Logi Analytics believes that any product manager should be successful. Every developer should be proud of their creation and they believe that every application should provide users with value. Applications are the face of your business today and it's how people interact with you. Logi can help you provide your users the best experience possible. Logi's embedded analytics platform makes it possible to create, update and brand your analytics so that they seamlessly integrate with your application.

Visit logianalytics.com/sedaily and see what's possible with Logi today. You can use Logi to create dashboards that fit into your application and give your users the analytics that they're looking for. Go to logianalytics.com/sedaily.

[END]