EPISODE 1003

[INTRODUCTION]

[00:00:00] JM: Distributed stream processing systems are used to read large volumes of data and perform operations across those data streams. These stream processing systems often build off of the MapReduce algorithm for collecting and aggregating large volumes of data, but instead of processing a calculation over a single large batch of data, like MapReduce does, these systems process data on an ongoing basis. There are so many different stream processing systems for this same use case. There's Storm, Spark, Flink, Heron, there's many others. Why is that?

When there seems to be such of a consolidation around the Hadoop MapReduce batch processing technology, why are there so many stream processing systems? One explanation is that aggregating the results of a continuous stream of data is a process that very much depends on time. At any given point in time, you can take a snapshot of the stream of data and any calculation based on that data is going to be out of date by the time that your calculation is finished. There's a latency when you start calculating something and when you finish calculating it, and there are other design decisions for a distributed stream processing system. What data do you keep in-memory? What do you keep on disk? How often to you snapshot your data to disk? What's the method for fault tolerance? What are the APIs for consuming and processing this data?

Maximilian Michels has worked on the Apache Flink and Apache Beam stream processing systems and currently works on data infrastructure at Lyft. Max joins the show to discuss the tradeoffs of different stream processing systems and his experiences in the world of data processing.

If you are curious about stream processing systems, we've done lots of episodes about them and you can find all of these past episodes by going to softwaredaily.com and searching for the technologies that we discuss or the companies that are mentioned, and you can also find all of these information in our mobile apps that contain all of our episodes. You can listen to all 1,500+ of our episodes in the apps. If there's ever a subject that you want to hear covered, you can

leave a comment on this episode or a different episode. You can send us a tweet @Software_Daily. We'd love to hear from you.

[SPONSOR MESSAGE]

[00:02:25] JM: Over the last few months, I've started hearing about Retool. Every business needs internal tools, but if we're being honest, I don't know of many engineers who really enjoy building internal tools. It can be hard to get engineering resources to build back-office applications and it's definitely hard to get engineers excited about maintaining those back-office applications. Companies like a Doordash, and Brex, and Amazon use Retool to build custom internal tools faster.

The idea is that internal tools mostly look the same. They're made out of tables, and dropdowns, and buttons, and text inputs. Retool gives you a drag-and-drop interface so engineers can build these internal UIs in hours, not days, and they can spend more time building features that customers will see. Retool connects to any database and API. For example, if you are pulling data from Postgres, you just write a SQL query. You drag a table on to the canvas.

If you want to try out Retool, you can go to retool.com/sedaily. That's R-E-T-O-O-L.com/sedaily, and you can even host Retool on-premise if you want to keep it ultra-secure. I've heard a lot of good things about Retool from engineers who I respect. So check it out at retool.com/sedaily.

[INTERVIEW]

[00:04:02] JM: Maximilian Michels, welcome to Software Engineering Daily.

[00:04:04] MM: Thank you very much for having me, Jeff. I'm proud to be here.

[00:04:08] JM: Well, I'm proud to have you. I want to talk to you about stream processing. You are something of an expert in that area, and when I started doing this podcast around 2015, there were many different streaming frameworks that we're seeing used. There was Storm, there was Spark streaming, there was Flink, there was Heron. There were several others. Why are there so many solutions for distributed stream processing?

[00:04:34] MM: Yeah. I think that's a really good question. Actually, when I go out and give a talk about Apache Flink or Apache Beam or whenever I see somebody else give a talk, that question usually comes up and it's a very good question because there are a lot of stream processing analytics frameworks out there, especially now in the open source.

I think the answer to this is that stream processing, although not like completely new technique to process data, has just in the recent years seen a huge uprise because we have the computing power, we have the data and just the applications, the demand for applications is there and that's why also demand for stream processing is there.

I think going back to or just enumerating some popular frameworks, I can explain why there are so many. First of all, Storm, is probably like the oldest one that is known in the open source. It really became popular in 2013, 2014 and it had been around much longer. I think like Nathan Marz is the name of the original creator and he had a company, BackType, and that company got acquired in 2011 by Twitter who obviously had a great interest in like streaming applications.

They kind of tinkered around with it internally and they figured it's sort of too limited because the execution model doesn't scale well enough. It doesn't provide to guarantees that they needed at scale. They figured they just keep the API the same, but they rebuilt the internal design of Storm, and that's how they developed Heron, which first was just an internal closed source software, but was then released around 2015.

Yeah. I think this is like a popular pattern. You see this also with, for example, Samza, Apache Samza, which was developed at LinkedIn I think around the same time. It served also like a special need for stream processing at LinkedIn and later was sourced. They had slightly different requirements. They also had Kafka coming up internally and they were more worried about exactly once guarantees and integrating with Hadoop's scheduler called Yarn.

When I started talking about this, there are a lot of details to this, but I'm just going to try to keep it short. Then there's Flink, which goes back – I did some research before. It goes actually back until like 2009 as a research project to Berlin.

[00:07:29] JM: Seriously?

[00:07:30] MM: Yeah. But nobody knew that back then. It was like a researcher project. But the company was created in 2014, which I was also part for the first two years and that obviously began the rise of Link, which I would say is probably the most sophisticated streaming framework and most complete in the open source. Maybe that's a bit biased, but I think there are good reasons for saying that. We can dive into that later.

It really embraced like streaming at core build into the runtime but still could handle batch datasets, which it was used also for in the beginning. Now everything shifted more towards streaming. Yeah, and there's Spark, or course. Spark was a Berkley project more focused on batch processing. I think Spark never really made it to stream processing to cover like everything that stream processing needs to do, because they kind of internally kept their RDD design, which is really elegant and perfect for batch, but is really hard to scale for streaming, because essentially it breaks down to having mini-batches which are for real-time streaming applications – I mean, they come with some drawbacks. Not to say you can't use it for streaming, but I think Flink has the more – I'm not getting into the technical detail. It has the more like native streaming approach, which if I'm being honest, for some applications, although that's being addressed at the moment, it's actually worse than Spark like batch processing. I think Spark has some advantages there, for sure.

Just to go and finish this circle here. Then there's Beam. Beam is a project I'm involved with and I really love. It came out of Google in 2014 as like Google Cloud SDK and it had like a different focus. It didn't want to replicate a complete runtime also because there was Google Cloud Dataflow and all the other stream processors already. The idea was more like to combine them all, to create a unified API and allow – Just use one framework to execute on all these execution engines that were already there in the open source. Then once that was there, the project figured it also needed to build multi-language support, which to that degree, I think has not been done by other projects yet. I think Beam is really interesting project in that sense.

Also, honorable mention to Apex from DataTorrent. I don't know if you came across it. Also, early on, I think they also got a lot of things right and had a really interesting framework for

stream processing. But unfortunately, they released it to open source and the company don't really exist anymore behind it. Yeah, I think that's more like history now.

[00:10:42] JM: Yeah. You've given a nice survey of all the different streaming frameworks and I think one way to describe – I think there's no easy answer for why there are so many of these things other than the fact that there are just – This is just a very big problem, and the idea of large distributed parallel processing over big datasets, it's such a heterogeneous problem that you're going to get a variety of different solutions because different companies are going to be looking and different open source contributors are going to be looking at this problem from different perspectives.

Now, I want to frame this conversation a little bit in terms of your own personal expertise and you've spent a lot of time in the Flink community. You've spent a lot of time in the Beam community. Let's start by zooming on Flink, and the Flink processing model. Flink keeps all of its state in-memory in order to have some low-latency processing and this system periodically snapshots the state of each of the nodes to disk. Can you give a more detailed description for what Flink keeps in-memory and what Flink snapshots to disk when it's doing its processing?

[00:11:53] MM: I think we have to differentiate there between what Flink keeps in-memory or in disk while it's processing and what is being written to disk or persisted when you take it to checkpoint. Those are two different things I would say, because when you have a streaming pipeline, I mean, the first problem you're dealing with before you want to ensure that it's exactly once and that it's checkpointed is that it's able to run.

I mean, if something crashes, then yeah, you want to be able to restore, but when you run, you have a data that is in-memory, obviously. Whenever you run any sort of application, it needs memory. What Flink has by default, it has the user code and the application logic in-memory. If you don't do anything – If you just want to have a normal Java program, you would just create your variables and everything would be memory. But Flink has like a dedicated interface that allows you to configure after you've written the pipeline how the state, the application state, is stored.

By default, to be more concrete, that is going to be in-memory. Everything is going to be in-memory. Obviously, you run into problems with that. Imagine you keep, let's say, a list of users or a map of users and session data in-memory and you have a million users. That could explode and consume way too much memory. The thing is you don't always need – Although, of course, you have to distribute a model of Flink where you have a large number of instances to scale your memory, but you don't always need all the memory.

What Flink actually has and what you can configure is called a state backend, and you can configure the state backend to store a flush memory to disk whenever necessary. This is implemented with RocksDB, but this is a technical detail. If somebody wants to look at it, RocksDB state backend. That basically is like a database. It's intelligently handling – Loading data into memory whenever it's needed and flushing back to disk when you haven't this user for a while.

This is the first step to achieving robust processing, stream processing, I'd say. Then the next part of your question is sort of what do I do to persist that? Flink has checkpointing built-in that is able to write all these state to priority that is necessary to restore the pipeline at any point in time to disk or to any other storage system that you want.

This is pretty tricky, and that's something that took a long time to get right, because obviously you could just hold your entire stream processing pipeline and then do the checkpoint or safe point of your memory. But that would cause a lot of backlog and latency. What Flink has developed is a way to asynchronously but also do it in a way that it produces deltas. So you don't always checkpoint the entire state, but you only checkpoint what actually changed. Flink has I think the most sophisticated stream processing semantics in that regard when it comes to handling memory and checkpointing.

[00:15:36] JM: It's worth discussing the differences between stream processing systems and data warehousing systems, because both of these systems are used for big in-memory calculations. I'd like to provide some contrast between stream processing and data warehousing. How are these types of systems used differently?

[00:15:54] MM: Yeah. Thank you. That's a very good question. I think stream processing and data warehousing are not fundamentally different, but data warehousing is a more complete framework for loading data, aggregating data, analyzing it and transforming it. Whereas stream processing can be a specific part of a data warehouse, but just tailored towards real-time data. The way I think data warehouses generally work is that they pull in data from various sources and stream processing could be or Kafka topic could be one source that you use, but then it's not usually used for analytics that are performed not real-time or near real-time, but usually with a couple of hours delay or it doesn't have the same guarantees that stream processing would provide. But I think I would understand stream processing to be just a part of a data warehouse in the sense that it is a source for real-time data. Yeah, that's how I see the both complement each other.

[SPONSOR MESSAGE]

[00:17:16] JM: The start of the new year is a great time to evaluate your career. It's a time to consider your salary. It's a time to consider your job, or maybe consider changing your entire career path. Seen by Indeed provides a path to new career opportunities while reducing the pain of the traditional job search process. Seen puts tech candidates in front of thousands of companies like GrubHub, Capital One and PayPal across more than 90 cities. Just create your profile from your resume and they'll match you to the right roles based on your needs. Every Seen candidate also gets free access to technical career coaching, resume reviews, mock interviews and even salary negotiation tips to seal the deal.

Join today and get a free resume review when you go to beseen.com/daily. That's B-E-S-E-E-N.com/daily, D-A-I-L-Y. Seen by Indeed is a tech-focused matching platform. If you're ready for a new job, you're ready for Seen by Indeed. Join today and you get a free resume review when you go to beseen.com/daily. That's B-E-S-E-E-N.com/D-A-I-L-Y.

Thanks to Seen by Indeed for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:18:46] JM: I want to talk through an example use case of stream processing. I think a common use case is something like clickstream data where you have a high volume of data, maybe it's coming into a Kafka cluster and you want to continuously ingest data from the Kafka cluster into a stream processing pipeline and then you want to perform some stream processing over it and then perhaps write it back to Kafka. Let's say it's an unbounded set of clickstream data coming in. Describe how Flink could be used to process that data.

[00:19:22] MM: What you describe is actually like the prime use case of Flink. When the stream processing APIs search emerged in Flink and people started using it, we put a lot of effort in the Flink community to make it work good with Kafka, because that was what everybody wanted to use.

In Flink – Flink itself has a very good support for Kafka both reading from Kafka and writing to Kafka. That's called like consumer-producer. I'm sure you're aware of it. Why is the support good? Well, when you create a connector for an external system, it's always error-prone because you have to know all the quirks of Kafka or any other system that you're trying to integrate with. Flink does a good job there, but it also is able to work with Kafka in the sense that Kafka is offset, an offset-based log. When Flink checkpoints, it can take note of the Kafka offset, and when you restore a pipeline, you start reading from that offset that was active during the checkpoint.

How would that look typically in a Flink application? You basically write – You instantiate the Kafka consumer with some properties and then you have a data stream that's the data type that you use in the Flink API, for example data stream of strings. Then you apply all your Flink processing logic, like map filtered, group by, and then you write back to Kafka. It's pretty straightforward. You can find it on the website, in the Flink website if you want to take a look at it.

One thing that Flink didn't have for a long time was integration with the Kafka producer to checkpoint to a Kafka topic. The exactly once guarantees will only sort of valid within Flink and when reading from Kafka, but not when producing to Kafka, but this has been addressed in newer Flink versions where also Kafka itself needed to find a way for systems or external systems to commit something to Kafka so that you really have an end-to-and exactly once

guarantee, which is really useful. If you have analytics process, you want to make sure that if you have a counter and you have to restore your pipeline due to a failure, you assure that that counter is correct and not off by one or off by many. That's something that Storm, for example, couldn't do and also Spark streaming initially couldn't do. I believe that fixed that with the structured streaming approach that is new in Spark. That's basically it.

[00:22:19] JM: I've heard of several people who use this kind of application. They use Flink for this kind of application where they're reading high-volumes of data coming off of Kafka and then they're doing some stream processing and writing it back to Kafka. One thing I don't understand is why wouldn't you use Kafka streams for that application? I guess, more generally, how do Kafka streams contrast with Flink?

[00:22:44] MM: That's a very good question. That also comes up often and it makes perfect sense, right? If you have a Kafka cluster and if you don't really have a lot of, let's say, compute intents applications, then why not use Kafka streams?

To be honest, I think Kafka streams is great because it's so easy to set up. Again, when you have a Kafka cluster, you basically just includes a library in your Java application and then you're off to doing streaming analytics without any need to set up the Flink cluster or another system to deploy that application.

I think that Kafka stream is a great library. However – And I think even the Kafka folks have pointed that out. If you're into compute state-heavy applications, then Flink is the more robust alternative. I've seen and heard from many Kafka users that when they do expensive computations, like lots of shuffles, network shuffles and when they have also – Yeah, compute-intensive calculations, timers, all that stuff that you want to do in a sophisticated stream pipeline, then Flink is usually more robust alternative. I'd say for simple applications, Kafka streams is perfect, and whether or not you need all the power of Flink, then depends how evolved your application gets.

[00:24:23] JM: Let's get into talking about Apache Beam, because that's what you've been working with more recently. Apache Beam is a data processing system that originated at

Google, and the goal is to provide a system that can do stream processing on top of a variety of stream processing engines. Can you describe the vision for Apache Beam?

[00:24:46] MM: The vision of Beam was not to create another open source framework and runtime that we have already so many of. Instead, the idea was to combine all the existing runtimes and have a unified, easy to use API to basically work with all of them. That was the initial idea.

In the beginning, when Google came out with Beam, the idea was just to do that for Java. Although always I think the idea was to expand to other languages, but it was just Java in the beginning and that was kind of convenient because all the other systems in the open source were written in Java.

The idea is really that you have a single API for both batch and streaming. Many systems have a separate API for batch and streaming. Basically, when you do the switch or you want to rewrite logic to be streaming, you have to use a different API. Beam wanted to fix that, so that's just one API. You use that API to write it and then you just basically provide a flag during execution saying, "Hey, I want this to be run on Flink or Google Cloud Dataflow or Spark," and then we ensure that the execution works the same on all of these by a very extensive set of tasks that we have inside Beam. They're called validatesRunner, and they basically check for all the features that Beam has and ensures that the runner supports it.

I should probably just tell you what a runner is. A runner is basically the part of Flink that does the translation. Beam, I mean. From Beam to the execution and then that you choose to run your Beam program with.

Just to complete the picture here. Later on, once we had all these execution engines supporting the Beam API, we figured we needed something more than Java. Yeah, Google came out with the Python API and supported that in Google Cloud Dataflow, and then it took now about two years to have full runtime support for supporting multiple languages. We have Python and Go now, and Java, of course, and we have managed to rewrite most of the runners so that they support all these languages. I think that's something that gets me really excited, because being able to use Python, for instance, unlocks a completely new set of applications, because we

have all these libraries, like NymPy, Pandas, TensorFlow, all these people who love to write Python, but they just hate Java or it's too complicated for them and they just want to get their work done. Now we have a Python API, which supports stream processing with Beam. I think that's one of the most exciting things in Beam today.

[00:27:56] JM: Talking more about the roots of Beam. Beam is associated with the Google Dataflow paper. Can you give an overview of some of the concepts of the Dataflow paper and how they impacted Beam?

[00:28:06] MM: Yeah. I think the Dataflow paper had a lot of impact on streaming semantics in general. Not only on Beam, but also on Flink. Back in 2014 when Google came out with Cloud Dataflow SDK, Flink was building or finalizing their streaming API. It was not so hundred percent clear what is stream processing even. Storm tried to do it and had very quirky API, let's say. I mean, most people don't find it very intuitive. It was not clear. When should you be able to do a stream pipeline? That Dataflow paper basically described the model or the semantics that a stream processing system should support. In that, there are things like parallel collections. How is data distributed when you process in the cluster? There is all the operations that Beam supports, like parallel do, which is basically the functionality where you put your code in Beam. It has support or describes how timers work. Basically, when you see an event, you can set a timer to be notified once a certain time has passed or an event arrives. It has watermarks which are an important concept that both Flink and Beam adapted. Maybe I should explain what watermarks is.

[00:29:42] JM: Sure. Yeah. Go ahead.

[00:29:43] MM: A watermark is sort of many people don't find it that intuitive, but I think it's not that hard to explain. In stream processing, when you read data and you want to associate time with that data, you could just take the current time that you're seeing on the machine, right? But when you have a set of machines, then they might not be in-sync with time. That's sort of inconvenient. Also, the time when your machine doesn't really have anything to do with the data itself, most data usually has a timestamp associated. So you want to be able to use that timestamp, and because time is kind of irrelative, when you process fast, if use processing time

or time on your computer, then if it's fast, then less time elapses. If it's slow, then time elapses slower from the application perspective.

What watermarks are, they are basically a time indicator in a streaming application. What that means is the watermark passes from the source where you ingest your data throughout the entire pipeline, and the sources basically – They send out these watermarks priority when they think they have seen all data or a particular time span that you're interested in.

I mean, theoretically, you could wait forever, because you could always receive some data at the latest timestamp, but the watermark basically tells you, "Okay. Now, I'm confident enough that the time is now. I've seen enough timestamps where my data is at 10 and now I'm sending out a 10." that kind of triggers the whole – That allows all the operators who might be holding back data waiting for timestamp 10 to then kickoff their execution.

Another concept that is in that paper, which I should mention, is windowing. That's basically the idea that you group your data according to time or number of elements. Usually it's time, and that's where the watermark also comes into play.

For example, early systems, like Storm. There was no notion of a window. You would just receive all your data and then you could do whatever with it. You could implement your own logic. But in Beam or Flink, we have windows which allow us to group data and allow for the execution to only kickoff when we know that we have received all the data for a particular time span for a particular user.

Windows, they're not only by time, but also by key. If the data is partitioned, then we also have by key. Let me give an example to explain what that means. Let's say I have a clickstream and I am trying to count how many users –How many links were clicked for every user. Let's say I want to do that and I want to do that for the last hour, let's say, or for last minute let's say. I would just receive all the click events that users produce from a Kafka topic, for instance, and then I would define a one minute window and use to user ID to partition the data. When I received the watermark and I just check, or Flink internally checks, that if one test passed and then I received all the events that were produced within that minute and I can calculate, I can just count them and admit the count for the user. Then, yeah, could have a nice dashboard tell

me, “This user clicked on most links,” or something like that. I could rank them. That's one example how windowing would work.

[00:33:45] JM: Do watermarks define the windows?

[00:33:47] MM: Watermarks are like a time indicator, and to give a better example there, I think to understand watermarks, you need to also understand event time versus processing time. The processing time is just the time you are seeing on your computer while you’re processing data. The event time is the time that is contained in your data itself. What the watermark does, because we don't have this natural notion of time progressing in event time, just not like a clock always progresses and processing time, because we need a way to tell what time it is, and that's what the watermark is.

The watermark is basically sent from the sources through the entire pipeline and it can be held back if we are currently doing processing in one of the operators. It might be held back for the downstream operator. It basically traverses through the pipeline and tells, for example, window computation when it has seen all the data for a window time span.

When I have a window from, let's say, 3 to 4PM, then when that watermark arise at 4PM, I know that I've seen one hour of data now, of event time data, and then I can kick-off the computation for that window. That’s basically what a watermark does.

The tricky part about watermarks is how do we generate them. That's probably the most problematic about the watermark, because we could just send like the watermark. The watermarks is just a time stamp. We could just send a watermark directly when we’ve seen a timestamp in our data, right? When I see an event that is from 4 PM, let's say, like in our example, I would just immediately forward it to the subsequent operators. But if I do that immediately, what about let's say events that I receive later because let’s say you have an application where the best example is always a gaming applications. You have it on your phone. You’re offline. You’re in the subway. You lose connection and then you get out and all these events are being sent.

You want to be able to deal with some SKU in the event time. What you typically do is that you define a threshold that you're comfortable with. You would say I accept an hour delay or something that. That's basically a way that you can have some out of orderness in your data.

[00:36:45] JM: Right. Yeah, I understand.

[SPONSOR MESSAGE]

[00:36:56] JM: When you start a business, you don't have much revenue. There isn't much accounting to manage, but as your business grows, your number of customers grows. It becomes harder to track your numbers. If you don't know your numbers, you don't know your business.

NetSuite is a cloud business system that saves you time and gets you organized. As your business grows, you need to start doing invoicing, and accounting, and customer relationship management. NetSuite is a complete business management software platform that handles sales, financing, and accounting, and orders, and HR. NetSuite gives you visibility into your business, helping you to control and grow your business.

NetSuite is offering a free guide, 7-key strategies to grow your profits at netsuite.com/sedaily. That's netsuite.com/sedaily. You can get a free guide on the 7-key strategies to grow your profits.

As your business grows, it can feel overwhelming. I know this from this experience. You have too many systems to manage. You've got spreadsheets, and accounting documents, and invoices, and many other things. That's why NetSuite brings these different business systems together. To learn how to get organized and get your free guide to 7-key strategies to grow your profits, go to netsuite.com/sedaily. That's NetSuite, N-E-T-S-U-I-T-E.com/sedaily.

[INTERVIEW CONTINUED]

[00:38:45] JM: Talking more about what Beam actually does. Beam is a system for being able to express a data pipeline and choose different underlying execution engines. Why would I want

to choose different underlying execution engines? Why would a data pipeline have different performance times on a different underlying execution engine like a Flink versus Spark or Storm?

[00:39:14] MM: That's a very good question. I think many people who sort of board into a single or like one execution engine ask themselves, “Why should I use Beam now? I'm fine with what I have. Flink works reasonably well. Why should I use Beam now?”

Well, I think for many people, the answer is, “Don’t use Beam. Just use Flink directly.” However, I mean Beam does offer a bit more flexibility. You mentioned already you can swap out the execution engines. For many people and decision-makers, that's like an interesting aspect, because let’s say you’re in a Google Cloud. You’re Beam, Google Cloud Dataflow, which supports Beam. You may be run into some issues there. Your client demands that you need to be on-prem. So then you can then just take the same code and run in the open source on any Flink cluster, for instance. That gives you flexibility.

Same also the other way around, if you maintain your own cluster, maybe you want to run a new pipeline, a new instance of a pipeline in Google Cloud because you have that Flink cluster, but you want to try out something new in the cloud, in Google Cloud. That's just an easier way then.

Basically, yeah, support for multiple execution engines is interesting for some people, not for all. Then we have the unified API, which I think is also compelling argument, because you just write once for batch and streaming and then you basically just need to flick a switch to run that in either batch or streaming and also to deploy it on a execution engine like I mentioned.

Probably the third reason is the multi-language support. I think that's probably the most compelling argument nowadays, because there is simply no really good solution for running Python, scaling Python pipelines like Beam offers at the moment.

[00:41:22] JM: You currently work at Lyft. Tell me about the data pipelines at Lyft.

[00:41:25] MM: Sure. I work with Lyft or started working with Lyft because I think it is really interesting what they're doing with both Beam and Flink. They’ve been a Flink user for several

years now. It was their way to build a real-time analytics system at Flink. For now, I think almost 2 years, they're also Beam users, which came naturally to them, and that ties into the previous question that why would you use Beam? They could, because they had already the Flink architecture in place and all the knowledge to run and maintain Flink. They could just add Beam with Python support on top and to enable a completely new use case.

You have to understand with Lyft as a Python shop, essentially. That's what everybody says. They're not a lot of – Let's say the majority of the people like to write Python rather than Java. Beam is just the perfect tool in that regard. Also, there is a lot of existing Python code which even if somebody wanted to port everything to Java, it would be really hard because of TensorFlow and all the computational libraries that you haven in Python.

Yeah, I think a lot of frameworks in the past try to implement Python support, but usually did it in a way that it would, for example, use Jython, like Java execution of Python code, which never really supported all the use case, because in Python you have these C libraries like TensorFlow and NumPy, which require, yeah, a native Python interpreter, the C Python interpreter. Yeah, Beam could do it. That's why Lyft started using Beam.

To be honest, I don't have full insight in all the different teams what they're doing. I'm not even sure if I'm allowed to talk about this, but I know Flink is used for any kind of real-time analytics. They use also Flink SQL. They have then various – Basically, build an easy way to do SQL applications to get real insights into the driving data, for example, all the data that Lyft collects. I can't really go into details there, but I can talk a little bit more about Beam.

The Beam layer at Lyft is responsible for the real-time price factor calculation, basically. Lyft has also given talks about this in the past. Basically, what we do is we get all this data at Lyft and to accurately predict them, basically, the price factor for a given region that somebody wants to go to or is ordering a Lyft from. We take this data. We do feature generation, model training and model execution. Basically, what many other companies do also these days. But it's pretty exciting for Lyft, because before they were operating on a Python legacy stack, which can really tell you too much, but is basically a pile of scripts which is deployed via Airflow. Beam is a more scalable way to deploy these price factor calculations and also enables real-time price

calculation, which before the model could be old and based on old data. Now we have real-time prediction of the price.

[00:45:16] JM: Cool. As we begin to wrap up, let's zoom out. Tell me about the other areas of the data engineering ecosystem that are interesting to you.

[00:45:25] MM: Yeah. I think I'm particularly interested in the Kubernetes community. I think what they've done is remarkable. A couple of years ago, there were technologies like Mesosphere, Mesosphere the company, and DC/OS stack product, which still exist today. It's a good product, for sure. There was Docker Swarm. Yeah, there's definitely a bit more competition. But Kubernetes seems to have late conquered them all. It's just this huge community and it's just something – I just find the tech impressive and exciting to see this becoming mainstream that every company has their Kubernetes cluster.

I think working with it, there's still a lot that could be easier to configure and to set up and it would be really interesting to see the community build a layer on top of Kubernetes where it's not just like some infrastructure tool, but I guess that goes into the direction of serverless. Make it really easy to deploy your functions, to define dependencies between functions and data and make it really easy to deploy that.

I think we see this already, but I want to see this become a bigger pattern. That's also something where I think Beam and Flink could improve, and I'm seeing actually both project doing something in that regard. Flink with SQL. Beam also has SQL, by the way, which works quite similar. That enables analysts who are not programmers to do analytics themselves. I think that's great. Flink has moved into the direction of stateful functions. Maybe you've heard about this. It's a kind of new way to write Flink applications where you can basically compose it out of functions that talk to each other, which is also I think it might be a better or more comprehensible way to write streaming applications in the future. We have to see.

Really, I think these days, being like an engineer that is involved a lot with architecture and hard to get around concepts for average users. I really wish using all these products would be much, much, much easier. I am really just sometimes ashamed how hard it is to use this stuff. When

you know it, it's not hart, but when I see people struggling with it, I always wish things would be easier. I think we can do a lot there to improve.

[00:48:08] JM: Max, thanks for coming on the show. It's been great talking to you.

[00:48:11] MM: Thank you Jeff for having me.

[END OF INTERVIEW]

[00:48:21] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens, and we don't like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what's the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate's backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. So I'm a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[END]

