Kris Jenkins: (00:00)

So you can use Apache Kafka as an event bus. Events get produced to it. They sit around there for some number of milliseconds or minutes or months, and then they get consumed out. But if that's all you are doing, I think you're selling Kafka a bit short because sooner or later, you're going to want to start processing those streams, transforming the data, and joining it and aggregating it.

Kris Jenkins: (00:25)

And there's actually a lot of support for doing that if you get into the world of stream processing. But stream processing is one of those things that can be hard and easier than you think. So in this episode, we're going to talk to Jeff Bean and Matthias Sax about different approaches to it, why it can be hard, and how things like KSQL and Flink can make it easier.

Kris Jenkins: (00:47)

And somewhere along the way, we end up discussing XML and Cobalt, too. Before we get into it, Streaming Audio is brought to you by Confluent Developer, which is our site that will teach you everything we know about Kafka and stream processing, including some step-by-step tutorials for some of the things we discussed in this episode.

Kris Jenkins: (01:06)

Check it out at developer.confluent.io. And if you want to start a Kafka cluster running for your own stream processing project, head to Confluent Cloud and register with the code PODCAST100. You'll get $100 of extra free credit and you'll be up and running in minutes.

Kris Jenkins: (01:23)

And with that, I'm your host, Kris Jenkins. This is Streaming Audio. Let's process some streams. My guests today are Jeff Bean, who is the technical marketing manager here at Confluent and a former evangelist of things, including Apache Flink for data artisans, and Matthias Sax who works on Confluent's KSQL project among other things. He's got a Ph.D. from the University of, where was it, Matthias?

Matthias J. Sax: (01:57)

Humboldt University in Berlin.

Kris Jenkins: (01:59)

Humboldt University in Berlin. Gentlemen welcome to the show.

Matthias J. Sax: (02:04)

Thank you.

Jeff Bean: (02:04)

Hi.

Kris Jenkins: (02:06)

You've both got a background in obviously Confluent KSQL, but also Apache Flink and some other Apache projects. Let's start from, let's go right back to basics and see if we can trench through the world of stream processing. My idea of stream processing, my naive idea is I've got Kafka running. This is great. I write a consumer. I write some code that starts chewing through a piece of data and spits out events on a producer. I'm done. That's stream processing. Right? Consumer, producer, a bit of processing. Why is that not the whole story? Why do we need anything more than that, Matthias?

Matthias J. Sax: (02:50)

Yeah, I mean, there are multiple things about that. And in particular, I would like to mention two aspects. One is stateful stream processing. If you have to manage state, then your program just becomes 100X more complicated and building this yourself is quite challenging. And the other one is Exactly-Once Processing. That's also very challenging. I mean, to produce and to consume and Kafka exposes low-level APIs, but it's also not straightforward to use them correctly in this setting. And so higher-level abstractions actually make this part much easier.

Kris Jenkins: (03:30)

Okay. Perhaps you should tackle those one at a time. And I'm scared of Exactly-Once Processing. So we'll leave that technical data until we get a bit later. Why is state harder? Give me an example of a stateful application, and tell me why it's harder?

Matthias J. Sax: (03:49)

Yeah. I mean, in general, every time you want to do something that is context-dependent and basically includes more than a single event, then you become stateful. That could be an aggregation over a time window, or maybe a global aggregation. Or if you join data, or if you just need to remember events from the past to make a decision in the future and a new event comes in to do a certain action.

Matthias J. Sax: (04:14)

And I mean, the problem is, of course, or this [inaudible 00:04:18] use a hash map or something like that. Put your Q value pairs in. But then of course, if you have any error, then you lose your state. You want to make the state for tolerant, but how do you do that? That's the main challenge. I mean, another naive approach would be, "Well, I write it to the local disk." But then, of course, you need to restart on the same computer to make it work.

Matthias J. Sax: (04:40)

Or you say, "I attach remote storage," then you can also use a different computer, for example, if you're in a more stateless environment, like Kubernetes or something like that where you deploy your stuff. But then you have a remote state, and you have more moving parts. What happens if your network connection to your low remote state goes away? Then you again, have a problem. Right? Then you need to take care. And then you basically come from one problem to the next problem, to the next problem. And it's just very, very complicated.

Jeff Bean: (05:08)

It usually works in such a way where you solve the immediate problem. And then you chug along for a while, and then you realize you have some higher-level problem or some next problem. Then it works. So a simple aggregation, maybe you don't need a whole lot of state for that. But now you want a rolling aggregation with a time window, or now you want to enrich data, by joining it to another stream.

Jeff Bean: (05:28)

And suddenly you have to code state in a way that's also like Matthias said, fault-tolerant, but also distributed and skill. And this becomes very difficult just in and of itself. And you start needing a whole framework for that. And then when you actually start relying on the framework, how do you keep the framework accessible and easy to adopt so that you don't have to learn the complexities of reasoning with time and building out different state back ends, and the strengths and weaknesses of each one and so on just to get your application out the door.

Kris Jenkins: (06:04)

Okay. You've both spent some time with solutions to that problem of the increasing complexity of your stream processing. Right? Give me a quick overview, and then we can dive into the depths. What are the different ways of solving this?

Matthias J. Sax: (06:22)

Want to [inaudible 00:06:22]?

Kris Jenkins: (06:23)

Who am I going to put on the spot?

Jeff Bean: (06:26)

It's a very wide question. You would use frameworks for this. Right? Are you talking about different ways of addressing the stream processing problem? I think when you're coming at it from the point of view of a developer, you want to use the best framework for the problem you're trying to solve.

Jeff Bean: (06:44)

Apache Flink is very attractive for that because it can do a lot. It's very capable and it has a lot of these constructs that allow you to reason about things like state, and things like time, and so on. But then when you come at it with another angle, which is maybe that of a new user or a new someone who's just gotten their arms around consuming and producing events with Kafka.

Jeff Bean: (07:11)

Now they want to do the next thing. Now they want to do some basic aggregation, and then they want to add a time window, but they want to do it without spending months learning a development framework. Then you start looking at SQL-based stream processing such as ksqlDB or Flink SQL.

Kris Jenkins: (07:27)

Right. Should we do the low-level version first? Yeah, let's go with that. Take me through how, what a Flink stream processing application looks like first. Teach me something about that.

Jeff Bean: (07:42)

Sure. Matthias, or do you want me to go, or do you want Matthias?

Matthias J. Sax: (07:45)

Oh, go ahead.

Jeff Bean: (07:50)

Sure. Okay. Flink is implemented in Java. There is a streaming API that's layered on top of Flink. Flink has multiple API abstractions. It's layered, upon layers, upon layers. At the highest level, there's a data stream API where you define the data stream, and then you define operations that you're performing on the data stream at the API level.

Jeff Bean: (08:10)

You open the stream. You open a connection to state as you process the stream. You store things in state. And then you can define different operations like Windows and so on. Those things get compiled into operators that run in the Flink engine. Very similar to a very similar distributed architecture like Apache Kafka.

Kris Jenkins: (08:35)

Okay. Well, see, I don't know Flink very well at all, but I know Kafka streams, which sound very similar. Matthias. Is it broadly in a seminar?

Matthias J. Sax: (08:45)

It's absolutely. I would say 80% of use cases and 90% use cases, you can pick the one or the other, doesn't literally make a difference.

Kris Jenkins: (08:54)

Well, that's interesting. What's the remaining 20%?

Matthias J. Sax: (08:57)

Yeah. So the main difference is really the deployment model from my point of view. Kafka streams are built as a library. You basically say I embedded into my regular Java application and embeds a compute in my Java application. A very natural pattern here is microservice architecture. You build a lot of small microservices and bet to compute, you're done. Flink on the other hand has a cluster model. There, you basically need to set up and deploy a Flink processing cluster. And then you basically write your job and you submit the job into the cluster.

Kris Jenkins: (09:33)

Okay.

Matthias J. Sax: (09:34)

And then it's running in the cluster. You basically have more like a [inaudible 00:09:37] processing model. And then, of course, that's similar to KSQL because KSQL, you would also say you deploy a bunch of ksqlDB services, and then you submit your SQL. So there are really the differences here between client-side processing or service-side processing if you wish.

Kris Jenkins: (09:53)

That must change the way you write your applications. Right? If it's a library, there's going to be more, I assume there'll be more mixing and matching between stream processing code and your regular application code.

Matthias J. Sax: (10:04)

Yes. That happens a lot, especially when you want to tap into your state. In Kafka Streams and ksqlDB, we both allow you to query a state a talk. In Kafka Streams, we call it interactive queries. In ksqlDB we call it poly queries. Do an aggregation, you build a table and then you can query the table directly. I think Flink had a similar thing at some points. They called it Critical State.

Jeff Bean: (10:30)

Critical State.

Matthias J. Sax: (10:31)

I'm not sure if that's still maintained. But then of course, here you would actually start to clear a remote state because your application is running server sites. The state is on the server. So it becomes a slightly different pattern.

Jeff Bean: (10:43)

And the state is also a different back end. Usually, it's RocksDB or something else. And you can swap it in and out, which is fine, except the behavior of the application changes depending on the state back end. Now the developer building the string processing application has to worry about how the administrator has deployed the back end.

Kris Jenkins: (11:02)

Oh, God.

Jeff Bean: (11:03)

And [inaudible 00:11:03] that out, then your application changes. It becomes a bit of a troubleshooting nightmare.

Kris Jenkins: (11:08)

Right. Hang on, step back a second. Where does RocksDB even come into this? Why do I need another database?

Matthias J. Sax: (11:14)

To some extent, it's an implementation detail, on the other hand, it's not. And I mean, it's a full tolerance mechanism in Flink and Kafka Streams, ksqlDB quite different. In Kafka Streams, you basically started out saying we don't want to have a lot of additional dependencies. And we do already have a Kafka cluster.

Matthias J. Sax: (11:33)

What we do to back upstate is we create an additional topic in the Kafka cluster that we call a change lock for the state. And every state modification is also written into this, to this topic as a writer headlock. And then if you lose the state, then we can replay the change lock topic from the Kafka cluster and we are fine. And because the cluster is here anyway, because it's our input and output so there are no additional dependencies.

Matthias J. Sax: (11:57)

Flink is choosing a different approach to basically say you configure a so-called state backend for tolerance storage that could be an S3 or an HDFS depending on your environment. And then what Flink is doing is basically saying, "Well, when we have those RocksDB files sets to a local state, we actively copy those RocksDB files into the remote store. And then when we need to recover, we can copy those files back."

Kris Jenkins: (12:25)

Okay. Is that done right ahead as well? Or have you've now got a transactional thing going on?

Matthias J. Sax: (12:30)

It's a right ahead thing as well. Flink also has a mechanism of committing. They would say, "I note how much input data I have processed in Kafka, committing offsets." And before you use this, you would basically say a first copy also files or in Kafka Streams ksqlDB first flash or pending rights into the change lock topic to make sure we don't lose anything. And only after that, we commit the offsets and that basically gives you at least months of processing semantics.

Kris Jenkins: (12:57)

Okay. So I see. Yeah. I was going to ask how it is that when it's a library on your, that you are running inside your app, it can actually do distributed storage. And the answer is to re-use Kafka, and write it to a topic. Right?

Matthias J. Sax: (13:11)

Right, exactly.

Kris Jenkins: (13:12)

Yeah. That makes sense.

Jeff Bean: (13:14)

That's the Kafka Streams answer. Or the Confluent answer is Kafka becomes the equivalent to the state backend. There are a couple of times when I was working with Apache Flink where this is really inadequate because the amount of traffic that is required for the stream processor to actually manage state and the number of reads and rights to the state backend start dwarfing the amount of just normal Kafka traffic.

Jeff Bean: (13:42)

In that case, now you start going to separate the back ends. I think of this more as an anomalous use case, or a special one rather than the default, which is Kafka's fine. Tort your intermediate data in Kafka and it's fine.

Kris Jenkins: (13:57)

Yeah. Okay. Yeah. I see how that throws together. That gives me a sense of how these things are operating on the back end. Jeff, is my life going to be seriously different apart from the deployment model as a programmer re-using these stream processors?

Jeff Bean: (14:20)

Re-using either Flink or Kafka Streams?

Kris Jenkins: (14:22)

Yeah. Yeah. How is my life going to change?

Jeff Bean: (14:25)

The framework is different. The APIs are different. The interfaces as a developer is completely different. I can't really arbitrarily switch from one stream processor to the other without rewriting my application. That's why it's critical to make a good decision early when you're building these things.

Jeff Bean: (14:44)

There are strengths and weaknesses to each. I would suggest that working within the same platform for your stream processing and your event streaming is a good place to start. Right? You don't want to begin consuming and producing with Kafka, come across a stream processing requirement, and now go learn an entirely separate distributed system, which is what adopting Flink would require.

Jeff Bean: (15:06)

A lot of Confluent customers actually are not doing stream processing yet. There's plenty of room to grow in the platform just to, without needing to go adopt something else.

Jeff Bean: (15:19)

In my view, you have to hit a brick wall of some sort before abandoning the platform altogether and going to some other stream processor. Flink has quite a learning curve on this. I was doing evangelism and education for Apache Flink for about eight months, three to five of those months was just learning the basics well enough to teach it to someone who knows absolutely nothing.

Jeff Bean: (15:42)

There's a significant investment in that when you select a technology, which is, I think not to be understated. The productivity you're going to have, and by your development teams and your data teams are going to continue to grow and your team is going to continue to be effective if you can stay within the same platform.

Kris Jenkins: (16:04)

Yeah. That leads us onto the motivation, the mental on ramp, the motivation for having an SQL-like interface. Right? Because to a certain degree, you can assume that all programmers know some SQL. But take me through that evolution from saying, "Okay, we've got it as a library. Now we're going to jump into supporting KSQL." How did that come into being? Jeff, you're the product guy.

Jeff Bean: (16:37)

Sure. It's a trend, you'll see in a lot of the stream processing space that's out there is you want to make it simpler. I've already dinged Flink a few times for being difficult to learn. You should not need a developer to do stream processing. Arguably you should be able to take a data analyst and take their existing skills and be able to process this stream much like they would issue an aggregate query against the database.

Jeff Bean: (17:08)

That's the dream. And that's kind of the requirement that we're all chasing in order to lower the bar on stream processing. Flink has Flink SQL, it's a project that compiles a SQL query into a Flink job and then deploys the job to a cluster. We have ksqlDB. They're similar on the surface. One of the things that ksqlDB offers, I think that makes it a cut above is the integration with Connect so that I can define a connector and a ksqlDB query, load that data from that connector into a Kafka topic, and then query it right away.

Jeff Bean: (17:46)

And then sync, define a sync connector and write the result out to something else. I have my whole end-to-end data pipeline in a single query or a series of queries. That's one of the things that I think ksqlDB benefits from, is I can do more faster. It gives me more value quicker.

Jeff Bean: (18:04)

I think on the Flink side, they've really spent a lot of time being able to express complex ideas. There is an API or a function or a series of functions in Flink SQL for complex event processing, where I can actually define a series of events by how they look and then process them as a chunk in a Flink SQL query, which is interesting. And they've also do-

Kris Jenkins: (18:29)

Can you an example of that? Sorry to interrupt, but make that concrete for me.

Jeff Bean: (18:34)

... Okay. Let's say I'm tracking trips, taxi trips. And a complete taxi trip might be a start event, a series of stop events, and then some final end event. Right. And then maybe at every stop event, there's a charging event. So I have a series of events that together have meaning, and I want to identify those together, even though they're spread across over time, and then process them as a unit.

Jeff Bean: (19:08)

I think you can do this in ksqlDB as well with the table streams duality, you can define tables and have each event type be a row in that table, and then query the tables together. But the way Flink has your reason about it is with this complex event processing language. That's one aspect of it. Again, it's this trade-off between complexity and ease of use and ease of adoption.

Kris Jenkins: (19:35)

Okay. Matthias, that brings me into the... So the great thing about adopting SQL as your front end, as your interface for stream processing is it's going to feel very familiar to people.

Matthias J. Sax: (19:49)

Yeah.

Kris Jenkins: (19:49)

But it's not the same. When I log into ksqlDB, I'm not getting Postgres. Let's do a survey of that. What different concepts are there under the hood, and what do I have to actually adapt to as a user?

Matthias J. Sax: (20:03)

Yeah, yeah, absolutely. I think since the main difference is really that SQL was built for static data if you wish. I mean, you have your tables and you in a talk manner, you issue a query. And at this point, sure, there might be concurrent transactions, but in the end, you query more or less runs on a snapshot of the database. And so you have a finite data set.

Matthias J. Sax: (20:26)

The query computes the result, returns the result to you, and terminates, and that's it. In stream processing, we want to do this quite different because it basically said, "Well, we want to have a standing query that is computing all the time." So you have potentially infinite data. And especially in ksqlDB, we go different routes than Flink. So we basically really say, you can... You have two objects, you not only have tables, you also have data streams.

Matthias J. Sax: (20:50)

You can say it creates a stream statement and it creates a table statement. And then when you query one or the other, you also get different semantics because a table is basically, it's a revolving set of records. It's a change over time-

Kris Jenkins: (21:03)

[Inaudible 00:21:03]-

Matthias J. Sax: (21:04)

... Like a database, but the query really tags along with the changes instead of executing over a snapshot. But then, on the other hand, you have this stream, and the stream is basically an immutable sequence of events. There are no updates. The semantics are different as well. And that's also why we exposed different operators. For example, if you join two tables, well, you just join two tables, and whenever input in one of the two tables shows the changes, then you just update the corresponding result. So straightforward, like you would expect it.

Matthias J. Sax: (21:37)

If you join two data streams, you can't really do that anymore because, well, data streams are infinite. Here's a language we basically say, no, we enforce you to limit the scope of the join using a joint window to basically say only join events that are in time close to each other. Because otherwise, you need to buffer everything. And input is infinite. In here, you see these different semantics shining through because when you say join stream one, two-stream two, we force you in SQL to say, specify the window. In a table, you don't have this clause. And then SQL starts to diverge. And you [inaudible 00:22:13] stand if you query a stream or a table because the query will behave quite differently.

Jeff Bean: (22:17)

One of the things I find really interesting about SQL-based stream processing, in general, is the limitations that people run into conceptually. And I have examples on both sides. When I was teaching Flink SQL, I was at a large, web-based company that was not to be named, the teaching of a Flink class. We were talking about Flink SQL.

Jeff Bean: (22:38)

And the person in the class had a Flink SQL job running. It was very slow. And we were able to go and use Flink techniques to go in and debug that job and figure out that the parallelism was not high enough. The solution was to increase the parallelism of the running Flink job.

Jeff Bean: (22:54)

You can't do that without killing the query and reissuing it and starting it over again. And that led to a downtime. There's that limitation on the Flink side with SQL-based stream processing. On the Confluent side, I was... One of the strengths of ksqlDB is being able to connect to other systems and also being able to have a nice front ends to it.

Jeff Bean: (23:16)

Any application that emits SQL, whether think of a business intelligence application or an analytics application of type can emit ksqlDB statements that work. And we have proof of concepts that work. We have partners that do this.

Jeff Bean: (23:29)

I was in a meeting with a product manager for an analytics company. And she basically asked she said, "I don't understand how this works though. If I emit ksqlDB queries and stream the result to my dashboard, what does the customer do? Do they watch it like a TV? And that's good screen processing? Or is it something else?" And they were looking to us for that answer. And we were looking to them for that answer. "We're giving you the sequel dates front end. And you get to do what you want." And-

Kris Jenkins: (23:59)

What do you fancy?

Jeff Bean: (24:00)

... Yeah. So I find it really interesting that we're trying to make things easier and easier to adopt and easier to reason about. And then we run into these kinds of, not exactly limitations, but what's the right answer to these kinds of questions? I think it's still to be determined.

Kris Jenkins: (24:15)

Yeah. I think you see that. I think you see that in functional programming. You need a different mental model for dealing with, in some cases, infinite lists of immutable data. And then you see it in front-end programming where, okay, you can poll an HTTP and endpoint and get a snapshot of the world.

Kris Jenkins: (24:36)

And that's one way that will influence the way your UI is designed. Or you can connect to a web socket and then you have to deal with this infinite stream of stuff happening to you. And it's not that they're technically a wildly difficult concepts, but they're so unfamiliar. You have to rewire the way you think about application building.

Matthias J. Sax: (24:55)

Yeah. Yeah. I totally agree with that. I mean, the point is really, it's a mental shift for developers, especially because if you're used to databases, just to have an idea of a query that doesn't terminate is kind of alien to most people. And the dashboard example from Jeff is just perfect. It's kind of, well, okay, it's streaming the data. So what do I do with it?

Matthias J. Sax: (25:18)

I mean, and that's the point. You need to re-architect your application from the ground up to basically integrate with stream processing. You can't just say, "Oh, I use Postgres and now I drop in ksqlDB and everything is just fine at the front." And no, you need to re-architect everything.

Kris Jenkins: (25:34)

Yeah. Yeah. And it's not like you need to re-architect it because you've switched library, you've switched mental models. You've switched the way you structure ideas.

Matthias J. Sax: (25:45)

Exactly.

Kris Jenkins: (25:45)

And I guess the primary payoff for this is the real-time aspect. Right?

Matthias J. Sax: (25:51)

Yeah. Also, yeah. And I mean, it's not just real-time. It's time in general. I mean, stream processing is basically always there's a time dimension to it. Right? And it's also a very new concept and it's hard to reason about. It's a whole different concept that I exposed. So Flink is using this watermark-based approach to reason about time, and Kafka Streams is using a so-called slack time approach with advantages and disadvantages to both.

Matthias J. Sax: (26:14)

But in the end, there's always time. And especially this concept of event time is quite new to a lot of people because most people if they look at the time, they think about wall clock time. Let's say you do a Window aggregation, then you say, "Okay, my input could have out-of-order data. The window en time is reached. I wait for another two minutes and then the window closes."

Matthias J. Sax: (26:35)

Well, that might not really happen if you do everything event time based, because in event time just don't want to wait for two minutes wall clock time. You want to wait for two minutes of event time to pass by before you close the window to give you 30 minutes to process. And then reason about this new event time concept and dealing with out-of-order data and event time and stuff like that, it can be quite challenging.

Matthias J. Sax: (26:58)

And there actually also is where SQL comes into place and helps a lot because we try to have higher-level abstractions where many of those things just work out of the box while the lower level programming you do, the more you need to take care of those things manually. Going back to your original example like, "Okay, I use a consumer and a producer and do my stream processing myself." Well, then you have to think about all those chronic cases.

Kris Jenkins: (27:23)

Yeah. So that's one way you can go and get your own Ph.D. Right?

Jeff Bean: (27:27)

It gets highly relativistic very quickly. You think you're reasoning about time in your consumer or your producer, but now you're realizing, "Okay, this event was generated on some device on some part of the planet. And the time on the wall clock for that is one thing. And then the time is that the device stamped on the event was another thing.

Jeff Bean: (27:49)

And then there's the time it got to Kafka. And then there's the time it gets to you. Right? And then you throw on Flink, and then there's the time it gets to Flink, or is some other stream processor. And it starts becoming very mind dizzying very quickly.

Jeff Bean: (28:03)

And it works against this whole idea of making stream processing accessible. I love Flink because it lets you reason about all this stuff. I can swap in time semantics with one line of code change. I can switch from system time to event time. But the ramifications of that and how you tell that that's correct is really on you. And that's painful.

Kris Jenkins: (28:27)

Yeah. Again, that reminds me of the kind of relational database world where you've got those different serialization guarantees, like read uncommitted and recommitted. And you've probably read about them and you probably just stick with the default because it's too much brain space to spend. Right.

Jeff Bean: (28:42)

Well, it depends on the kind of developer you are, right? I mean, maybe you think stick to the default or you think this is most definitely not going to work for me. I have to build my own time management system. In the very short time I was working with customers using Apache Flink, I ran into two developers who decided to build their own watermark extractors because they didn't understand what Flink was doing so they thought they had to write it themselves.

Jeff Bean: (29:08)

They were both doing it wrong. I didn't understand exactly how they were doing it wrong. I had to go bring in a Flink committer, who had to tell me that they were doing it wrong. And then which, and then he was able to tell me how to use it correctly. Again, it's one of those things where when you're given too many tools in a space that's hard to reason about like this, it becomes difficult to be effective.

Kris Jenkins: (29:30)

Yeah. Matthias, I'm in the mood for some gory details now. Let's hear a little bit behind the hood. And maybe by the end of the explanation, I'll be not so in the mood for gory details and I'll just go with SQL. Different types of time-related joins, one I find interesting is a session join. I want you to explain what that's actually doing. Let's peer a bit under the hood so I can see how complex this soup is. Right? Session-based window joins.

Matthias J. Sax: (30:04)

Well, I mean, joins don't really have sessions in our API. I'm not sure if Flink has anything like that. I mean, we have session windows for aggregations. And there's the idea is to say that it's a little bit similar to the taxi example Jeff gave before but still different. You basically say, "I track the data. And if events are close to each other in time, I put them into the same session and process them together."

Matthias J. Sax: (30:35)

And then if there is a gap of inactivity, then I actually say, I split those data into two different sessions. For example, if you have a user logging in into a website, then you could say, "Now, if the user becomes inactive on the website for, let's say 10 minutes and doesn't do any interaction, then basically the user session ended."

Matthias J. Sax: (30:53)

And then I take all events inside the session for this user and I can do something with it. And then if the user comes back, it just starts a new session for the same user. You have to basically the user-based split out for you to process each user individually, and then you apply. So session detections basically based on time.

Matthias J. Sax: (31:12)

If you join, you just say, I have two input streams and I just have in the end, an additional predicate that tells me how far in time those two records from the left and the right input could be apart from each other. And if yes, it's part of the joint conditions they're allowed to join, plus an additional actor joint condition on the data you want to use. So you have a time condition and a data condition, and then you put them together.

Jeff Bean: (31:38)

One of the things that I think is just to add onto the example that Matthias just mentioned is you determine the... You capture a bunch of events from the first session. The first session closes. The user comes back, you're having the second session. But then the stream processor gets an event from the first session because things are late and things are out of order. And now, it's up to you as the application developer to figure out where to put this, do you discard it?

Jeff Bean: (32:01)

And each framework will behave differently in this respect. One of the things Flink can do is it can identify that second event or that later event from the first session, re-aggregate the whole first session and remit everything. But now as the application developer, you have to understand that it did that and understand what that means.

Kris Jenkins: (32:21)

What's the KSQL answer to that question?

Matthias J. Sax: (32:24)

Yeah, it's very similar. I mean, in general, ksqlDb is a little bit of a different model when it comes to admitting data. Flink is using this watermark approach. Right? So you basically say, "I hold off to send any output until I see a watermark." As example, Flink, if you basically say, "I have the session. I accumulate data, but I wait for the future. And if it's out of or, even [inaudible 00:32:48] add it to the session before I actually trigger the computation."

Matthias J. Sax: (32:51)

And then of course, in Flink you can also, after the fact actually re-trigger the computation if you keep the state around. And so ksqlDB uses a different model. ksqlDB basically says, "Well, every input is basically a change to your result." And so the output is also modeled differently.

Matthias J. Sax: (33:08)

In Flink, if you do an aggregation, you always get a table and the table basically contains a row per session in this case. And then every time an input record comes in, we identify the corresponding session based on its time and user ID, for example. And then we just update the row. You basically have this kind of state that is updated continuously in ksqlDB as a table, while Flink would say, "No, we accumulate all the events. In the end, we give you one final result and send it downstream."

Kris Jenkins: (33:38)

That must have latency implications. Right?

Matthias J. Sax: (33:41)

Yeah, yeah, absolutely. But I mean that's the way it is. If you say, I want to have a single final result, then you just need to wait until you can make a decision to say, "Okay, I'm done. I have seen everything." And because that is very hard to answer, we follow this different approach and say, "Well, we just incrementally maintain your result. And then as an application developer, when you let's say query the table, you make a decision when you want to receive the event."

Matthias J. Sax: (34:06)

And then you need to reason, is this current result fresh enough for me, yes or no? Why do I actually wait longer to query the data? But we just incrementally maintain the result for you and basically make it available to you. And it's up to you to use it or to wait and use only a more recent one later on.

Kris Jenkins: (34:24)

Yeah. So you-

Matthias J. Sax: (34:25)

But the problem doesn't go away, it just shifts from one place to another place.

Kris Jenkins: (34:29)

... Yeah. Yeah. It's whose pain it is. You're in this situation where you've got the live state in a KSQL world, but you have to accept that the client may reconnect later and things will change.

Matthias J. Sax: (34:44)

... in ksqlDB you can also have a cutoff point where at some point you say, "I don't accept any changes anymore." And then those new input events that are out of order would just be dropped on the floor, and you basically freeze your state and say, "No, that's it. I'm done."

Kris Jenkins: (35:01)

And are these all things you can say in KSQL? All these different settings?

Matthias J. Sax: (35:05)

Yeah, exactly. In KSQL, we express this as a grace period clause. When you do aggregation, you specify the size of the window or for a session windows, the gap parameter of inactivity. And then there's an additional grace period clause that tells you how long you are willing to accept updates to the state. And after stream time is passing the grace period, basically window and plus grace period, then you basically just freeze the window and its [inaudible 00:35:34].

Kris Jenkins: (35:35)

Yep. Yep. Like going to the cinema, you've got a start time and you've got to get in before the trailers finish. And if you're not there when the film starts, it's too late. Right?

Matthias J. Sax: (35:46)

Exactly.

Kris Jenkins: (35:47)

That's a bit of a stretch of a metaphor, but I'm going with it. That takes me into thinking, here you are, you've decided to put an SQL interface on stream processing. I'm sold. Good idea. And you've got a lot of syntax that you're just going to reuse. Right? But maybe this is a question for Jeff. You've then got a product design piece where you're taking a familiar syntax and adding to it, and adding new concepts, and trying to teach people in a feeling familiar way. How did you go about the whole process of that to use, developer experience, design for SQL plus?

Jeff Bean: (36:30)

It's a good question. I think the first thing you want does is be able to handle the basics. Right? So I want to be able to define most, if not all the use cases for my stream processor in my higher-level language. I want to be able to create a stream or a table, have those constructs defined for me, and define how to populate that stream or that table.

Jeff Bean: (36:50)

I want to have all my most common aggregations. I want to be able to find a time window and then declare the different types of time windows. And I want to be able to define queries on that. I want to have derived queries. This is much how a SQL works. And then there's room for new stuff, and how I define this new stuff I think is also interesting.

Jeff Bean: (37:15)

You can do things, you can move the language along quickly so that you can get easier adoption, or you can go for the standards-based legitimate kind of approach where you try to actually revise and evolve the SQL standard so that anyone who does NC SQL version X actually has a stream processing construct.

Jeff Bean: (37:41)

That one, I think takes a lot longer. And I don't think everyone who cares about SQL cares about stream processing. This is another place where you see the two technologies diverge a little bit. Confluent ksqlDB, the language itself is not concerning itself so much with evolving the SQL standard.

Jeff Bean: (38:00)

Whereas on the Flink side with Flink SQL and Apache Calcite and others, they've really have dug in on this whole standards question of really... They're starting to write academic papers on revising SQL for stream processing and all of this.

Jeff Bean: (38:11)

And I think that's interesting and great. Who knows how long it will be until we have enough consensus that you can have this be standard from technology to technology? I'm sure at the end of the day, what you'd love to be able to do is use the same SQL-based language and swap out the back ends based on whatever you need. I don't think we're there yet.

Kris Jenkins: (38:34)

Yeah. I'm not convinced we ever will be. I don't know if it's still a thing, but for a while, there was this dream in Java that you would write one interface file and just swap out Postgres or Oracle from my SQL. And it never works because you always pick your database based on the features of that database.

Jeff Bean: (38:54)

Yeah. And there's this-

Kris Jenkins: (38:56)

There's always something that you want that you can't get rid of.

Jeff Bean: (38:59)

... I heard an executive say something once, like when you're clearly in the lead, you don't care about the standards. And all the followers are going to try to use the standards question against you. I think this is an indication that Confluent is clearly in the lead in terms of ksqlDB and SQL-based stream processing.

Jeff Bean: (39:16)

But another thing I heard actually a Flink committer say, and this was in reference to Apache Beam, which is another abstraction layer for stream processing that's API-based. One of the critiques that he had, which I think also applies to SQL-based stream processing is one, if the abstraction layer isn't working for you, the first thing you do is get rid of it. And you actually start getting back to the brass tack. It's interesting in that regard, also.

Kris Jenkins: (39:47)

Yeah. Okay. That raises a difficult question then that I have to ask. Matthias, this is probably a good one for you to field. Right. So where are those holes? I mean, I assume ksqlDB is still evolving. Where are the gaps in the road where you would have to dip down into something like Kafka Streams?

Matthias J. Sax: (40:06)

Yeah. There are a few things. I mean, very often you want to be a little bit more imperative. I mean, SQL is a great declarative language. But with higher-level abstractions, there's usually, I mean, it's easy to use but there are also more limitations. Right?

Matthias J. Sax: (40:25)

You cannot express everything. The one good thing is really, if you want to do something on a periodic time scale, either event time or wall clock time. Both links support this. They have a so-called timer API. And Kafka Streams supports this as well as punctuations where you just say, "I registered this callback, and I want to re-execute this call back let's say every five seconds in wall clock time, or every 20 minutes in stream time. And then I want to do something, modify my state, clean up my state, or stuff like that."

Matthias J. Sax: (41:01)

That's a concept you don't have in SQL at all because there is a callback you can register. And what would it even mean? And how would you embed it in the language? Because at the end, you say, "Select star from whatever." It just doesn't fit the model. That is, I think one of those things.

Matthias J. Sax: (41:17)

And another thing is also, I mean, usually you also have limitations, what kind of data format you support. ksqlDB supports Afro and JSON, and then of course the Native Kafka, but then some people say, "Hey, I'm still from the '90s. I have XML data." And ksqlDB is just like, "Well, sorry, I can't parse that. We haven't built it."

Matthias J. Sax: (41:38)

And then when you fall down to lower-level APIs, you can always do this. Or if you have your own, whatever via format or whatever, then it's much easier to integrate. Teaching a database, also the formats is much more challenging. Also, when it comes down to data types and things like that, you need to define those translations because SQL has its own type system. There is an integer and a [inaudible 00:42:01] and a big end and whatever. And then it's how do you even translate between those different data types?

Jeff Bean: (42:08)

Matthias, you mentioned, that I'm the user who's still in the '90s. When I was doing partner work with Confluent, we were doing mainframe integration. I had a customer say, "Well, I'm still in the '70s and I have Cobalt copybook data that I want to ingest [inaudible 00:42:22] data with Kafka." This, it becomes again, really important to be able to work within the same framework. I can at least bolt a connector on that, ingest my whatever wacky data format I have in the Kafka, and then use a stream processor or a single message transformer, something to get that into some uniform process that makes sense.

Kris Jenkins: (42:42)

Yeah. Yeah. There's somewhere out there, there's someone writing the connector for soap. Right? The flavor of XML was even more of a truce.

Matthias J. Sax: (42:56)

Yeah, yeah. Absolutely. I think there are really the main challenges here. And once in a while, it's also just easier to express some very custom logic. I mean, ksqlDB also of course supports UDFs. That's very helpful and can close the gap here, too, in a lot of cases.

Matthias J. Sax: (43:13)

But once in a while, if you roll your own stream processor, especially if you want to really more complex state manipulations where it's not just aggregations and joints. But you basically say, "For every input record, I want to do some custom logic based on what I have seen before and what is in my current state." And then it modifies the state, and then it also triggers some output events.

Matthias J. Sax: (43:35)

You have kind of a, you could say a state machine model. Right? Implementing this in a channel function, that you just have a call back for every input record. And you get your state and you just modify it. It's much easier than expressing something like that in SQL.

Kris Jenkins: (43:49)

Yeah. Do you think it's coming? Are there plans to do that sort of thing on the SQL side?

Jeff Bean: (43:54)

I'm not convinced that it's not doable. I think there is a way to model, and use this stream table duality to model a table that represents your callback architecture. And then query that and show... I think it's possible, it's just not as intuitive. And in the, using the taxi trip example again, if I am looking to run a query for very, very long trips. Maybe taxi trips are longer than six hours, it's very intuitive to do that with a timer model.

Jeff Bean: (44:27)

Every taxi trip starts, I start a timer, and then every tax trip closed, I close the timer. And then if the time the timer goes off at six hours, and there's my report. Right? I'm not convinced you can't do it. I just think it doesn't fit how we think as easily.

Matthias J. Sax: (44:45)

Yeah. Yeah. I tend to agree to that. I think actually SQL itself is like touring complete if I remember correctly. But do you really want to use it this bad? I'm not sure.

Kris Jenkins: (44:57)

I don't know if you've heard of this, there's an annual coding challenge called Advent of Code. That happens around Christmas time every year. And an old friend of mine used to try, and so it's 24 puzzles in the run up to Christmas. And he used to try and solve every single puzzle just using SQL, which is phenomenally difficult. And most days he managed it. I believe it's true and complete. I also believe you don't want to do it for anything other than-

Matthias J. Sax: (45:22)

Yeah, I think so. But it's [inaudible 00:45:24]. It becomes not intuitive at some point. I mean, if I can just once in a while write a four-loop [inaudible 00:45:29] loop, I just know what's going on. Instead of writing five different sub-queries with correlated joints. And in the end, it's what is this query even doing?

Kris Jenkins: (45:39)

Yeah. Yeah. I think, well, there's a big missing piece that we said we were going to cover at the start that I have to ask you about the whole, exactly one semantics question that we punted that you don't want to implement yourself. You want your stream processing library to do. Matthias, take me through some of that.

Matthias J. Sax: (46:01)

Yeah. I think one aspect is against state. I mean, if you do exactly-once processing, you also need to include your state. If you modify your state during processing and something goes wrong, then you need to be able to roll it back.

Matthias J. Sax: (46:16)

In terms of questions, how do you use this? And especially when we integrate with Kafka here, so Kafka does have transactions and those transactions are designed in a way to say, "Well, I basically have an atomic operation that allows me to publish my result records into the output topics, and also record the input of said changes and makes this an atomic operation." So either the offset and the output both make progress, or both don't make progress.

Kris Jenkins: (46:46)

Now that seems like something-

Matthias J. Sax: (46:47)

[inaudible 00:46:47].

Kris Jenkins: (46:48)

... Sorry.

Matthias J. Sax: (46:48)

Right. And you also want to maintain the state. But the state is now an external dependency. Even if you use RocksDB, we have this external dependency. And then you basically, "Well, I could use transactions on my state back end." Sure. But then you have two transactions going on, so it's not atomic anymore. And then things start to become very complicated.

Kris Jenkins: (47:07)

Then you're reinventing two-phase commit just in your one-state processing app.

Matthias J. Sax: (47:11)

Something like that. Exactly. And I don't think you want to build a two-phase commit protocol yourself.

Kris Jenkins: (47:15)

I neither want to build nor use one.

Jeff Bean: (47:18)

I think Kafka has guarantees in this regard as well in terms of if I publish data on the topic, I know that any consumer can access that data and they're going to get the number of events that were published on the topic. They're not going to get zero events or duplicate events if I've published that, and that guarantee is really critical and makes stream processing easier.

Jeff Bean: (47:39)

In Flink land, if I'm reading from a Kafka connector, I get all those guarantees that Kafka brings. But now if I'm reading from some other connector, I lose them. And now I'm joining two streams in Flink, one of them has these guarantees, the other one doesn't. And now it's on Flink to ensure that this is end-to-end exactly once. There are capabilities in there for that, but it's painful.

Kris Jenkins: (48:01)

Okay.

Matthias J. Sax: (48:01)

Yeah, absolutely. Yeah. I mean, that's the advantage of Kafka Streams, and ksqlDB deeply integrates with Kafka so you don't have to worry about other things because your control the full ecosystem. And in Flink, you integrate with everything, and then it's, well, what do you do?

Kris Jenkins: (48:15)

Yeah. Let's just step back to finish up. Jeff, this is probably a question for you. Do you think there are businesses, customers, or places where one is a clear choice over the other? Are there certain kinds of use cases where you'd say, "Okay, that's one we should take with Flink," or, "This is clearly only going to work in ksqlDB"? Or is it more a taste thing?

Jeff Bean: (48:44)

I think it's a gradient thing. I think you can look and see who's adopting it. Right? And that gives you a clue. Most Confluent customers doing stream processing can do it within the context of

Confluent. A lot of Confluent customers are not doing stream processing yet and they, 90% of the time will continue to do it within the context of Confluent.

Jeff Bean: (49:08)

The outliers are the interesting ones, and they tend to be web properties with supermassive scale, and super big state, doing crazy aggregations. I know Netflix uses Flink for sessionization, all their session kind of stuff. I know Epic Games uses Flink as well for a lot of stream processing on their back end, even though they're Confluent customers.

Jeff Bean: (49:29)

I think there is a developer-heavy, very custom aggregation at supermassive state and high scale where Flink starts to look better. But you don't really know you're there until you're deep, deep in stream processing territory. And I think most of the time you're not going to get there. That's my take. I don't know, Matthias, if you have a different one.

Matthias J. Sax: (49:53)

No. No, I agree. I mean, Flink can be a little bit more efficient the way it's built, but it also puts much more burden on the developer to actually get to that level. If you take the Netflix and an Epic Game, and I think Uber, they basically have stream processing teams. They only do stream processing. And those teams become stream processing experts.

Matthias J. Sax: (50:16)

And that is the reason why they can leverage Flink in that way. If you're an average developer, you might not have the skillset to do that. And then using something like ksqlDB is just simpler or Kafka streams. And usually, you don't need to scale anyway, because it really is outliers that have this massive amount of data.

Matthias J. Sax: (50:36)

For most people, even if you use Kafka Streams, it basically idols around, it's, "Yeah, whatever. Give your events. I'm bored anyway." Because it's still high performance. I mean, Flink can still be 10X more, but if you only use 1% of Kafka Streams, then you would only use zero, not 1% of Flink. So what do you gain by using Flink? You don't need it.

Kris Jenkins: (50:57)

Yeah. Sometimes it's the complexity of the transformation, not the throughput of it. Right?

Matthias J. Sax: (51:01)

Yeah, exactly. Yeah. And then of course, in the end, I mean, it's also a question, how you want to go around and deploy? For example, ksqlDB, I mean, we offer it at Confluent as a fully managed service. There's no really such a thing for Flink. And then it's, you get support and you don't worry about it. You don't need to worry about the deployment. Those things are also very important to consider, obviously, if you just want to get your job done and not become a deep stream processing expert.

Kris Jenkins: (51:28)

Yeah. I don't think anyone wants to before the day they need to. Do they? Maybe. I don't know. But you get this. I perhaps have a final thought. I was building something recently with Confluent cloud and you get this, to me, it still feels strange. I go and look in the console and I've got insert into BLAS and a select query. I've got insert select from. And that's just running in the background. And I still find this a strange new world in some way, that insert is select isn't a one-shot thing. It's like this running process.

Jeff Bean: (52:03)

I don't trust it. Being at Confluent has had to teach me to trust these things. I'm used to, Tim Berglund did a video when he was still with us on the guy who needed to have his hands on the bare metal and really introspect. It was a fantastic video. You should look it up.

Jeff Bean: (52:19)

I wasn't that far along, but I do like to tail a log file. Right. And I want to be able to tail the log file connector and tail the log file of the running query. And then see if the hardware is getting in the way somehow. And trusting that this is just working and removing my grip from that log file so that I can just look at the data as streaming off of a ksqlDB query is shown in Confluent cloud is, it's scary, but it seems to work so far. And I'm pretty happy with it.

Kris Jenkins: (52:45)

Yeah.

Matthias J. Sax: (52:45)

And to be fair, the most use cases we still have is where you have an upstream interest. And then you process with ksqlDB so that you have a client-side interest. I mean, we have the capabilities to do that, but I think that this is I think very good for demoing and stuff like that. I'm not sure if anybody's using it in production at this point.

Matthias J. Sax: (53:07)

I think the good thing we have, is you see a lot of use cases in productions that actually query the state. So you have your persistent query running in the background, it's maintaining a table, and then clients are coming along to do look-ups into those tables. That is a very common use case where the client interaction comes into play with ksqlDB back end. But it's more on the read side than on the right side for production use cases at the moment.

Kris Jenkins: (53:32)

Yeah. I've played around with some interesting stuff like building dashboards where a lot of the logic of the dashboard is just a table select, rolling through that stream of events, building up a table. And then you can finally write your app as I'm just going to select from that table at the end, because all the state's there and processed.

Matthias J. Sax: (53:52)

Yeah, yeah, exactly. Exactly.

Kris Jenkins: (53:54)

Well, I think we should leave it there. And I'll part with some advice to Jeff, if you ever get that hankering to tail a log file, all you need to do is select star from stream omit changes. That will feel exactly the same.

Jeff Bean: (54:08)

Thank you.

Kris Jenkins: (54:09)

Thanks very much, Jeff Bean, Matthias Sax. Thank you for joining us.

Matthias J. Sax: (54:13)

Thanks for having us.