EPISODE 876

[INTRODUCTION]

[00:00:00] JM: The Hadoop ecosystem provided every company with the tools to store and query large amounts of data at a low cost. Since 2005, that ecosystem has expanded with more and more open source applications for data infrastructure. Data infrastructure includes databases, data lakes, distributed queues, data warehouses, query engines, web applications, on-prem software, close source, open source, cloud platforms and software as a service. There is so much software in the data engineering world and it is difficult to keep track of everything in the data engineering ecosystem.

Tobias Macey is the host of the Data Engineering Podcast, a show about technologies and practices in the world of data engineering. Tobias joins the show to discuss his perspective on the evolving landscape of data engineering and the trends he's seeing on the Data Engineering Podcast.

[SPONSOR MESSAGE]

[00:01:06] 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.

[INTERVIEW]

[00:03:25] JM: Tobias Macey, welcome back to Software Engineering Daily.

[00:03:28] TM: Thanks for having me, Jeff. How are you today?

[00:03:31] JM: I'm doing great. I'm looking forward to talking about data engineering with you. There are a set of components in a typical data engineering stack. There are databases, data warehouses, data lakes, queues. For someone who is just getting into data engineering, describe the different components of a data engineering stack.

[00:03:54] TM: Sure. So, I mean, they vary a little bit depending on the problem demand that you're working in or the type of organization, but to generalize, there are source systems. So, those could be application databases. They could be click stream data, like what you track in Google Analytics. They could be events from things like IoT sensors. But at the end of the day, you have some place where you're getting data from, and then you need to be able to eventually deliver it to somebody who's going to do something with the data. Whether use it for training a machine learning algorithm. Feeding it into a prediction engine, powering a business intelligence dashboard, and all of the steps in between are what we typically think of as data engineering.

So, from the source system, whether it's an application database or click stream analytics, you would typically either put that into a queue in the event of an unbounded stream of data such as events or run it through some sort of ETL or extract transform load pipeline. Nowadays, they're more commonly referred to as a workflow engine that you would use to extract the data from that source database and then load it into, whether it's a data lake or a data warehouse. Along the way, you also need to think about things like encryption and protection and access control of the data. Generally, particularly for data lakes, but also for data warehouses, you want to have some sort of a data catalogue so you know where the data is and what it is. Then, also, you need to be able to have some way of granting access to the end users who need to be able to access it. So you need to have some sort of identity management particularly for things like data warehouses that somebody has access credentials to be able to login and run queries to do the analysis that they need to do.

[00:05:44] JM: Why do we need so many different storage systems? We have a database, a data warehouse, a data lake. Why can't we just use one database or data system for everything?

[00:05:58] TM: Well, because different databases are architected to be optimized for different usage patterns. So, in an application, there are typically what are called OLTP or online transaction processing databases. So, those are optimized for being able to do quick reads and writes and being able to do some limited amount of aggregation so that the application can store and retrieve data in a fairly real-time fashion. Whereas for data warehouses or data lakes, we are going to be processing much larger volumes of data. You want to optimize for the analytical workload where you're aggregating information across thousands or millions or billions of records. So, a lot of those are generally column-oriented versus row oriented and things Postgres or MySQL. Then are also database such as MongoDB, which is optimized for being able to denormalize the data or store it all in a single object, which makes it easier to think about from the application perspective, but also has different attributes in terms of how you would store and access the data.

Then, in data lakes, you might have different ways that the data is formatted. So, it could be using things like Parquet, which is a column-oriented file format. It could be JSON. It could be

CSV. So, there are different types of data. Different ways that you want to be able to use the data, and there isn't really a way to be able to optimize for all of those use cases within one engine without having some sacrifice in terms of performance or throughput capabilities.

[00:07:33] JM: There are these different storage tiers with different access latencies and different costs. How does the tradeoff between time and financial cost? How does that affect the data pipeline?

[00:07:51] TM: So, in terms of the speed of access, generally, you want to – So, for more recent data, you want to be able to access it quickly in the event that you're working with a real-time application. So, things like sensor data where you want to be able to maybe do anomaly detection to see if there's some sort of significant event that you need to alert on, or if you're trying to monitor your production systems and you want to be able to generate an alert because all of a sudden you have a major spike in inbound web traffic and you need to be able to allocate more servers to be able to handle the load.

Whereas for stuff that is further back in history, because the time aspect is already so much further out, you're usually able to handle a longer latency in terms of being able to retrieve and process the data. So, just because of the temporal aspect of the data itself, it usually has some correlation with the allowances for how fast you want to be able to access it. Then in terms of the cost factor, being able to reduce the latency and the speed at which you can process the data usually means that you're working with solid state drives and high-throughput network access. Whereas data that you're able to accept some latencies, you might put that on spinning disk, or depending on how old it is, you might be able to put it on tape archive and deal with retrieval time on the order of things like hours or days versus the seconds or milliseconds that you might want for more recent data.

So, the faster data access costs more just because you're dealing with more high-performance hardware, which itself costs more for the person who's providing it versus things like tape archive, which is fairly cheap and ubiquitous.

[00:09:41] JM: Let’s imagine a ridesharing company. How does data make it into these different areas of the data pipeline? Take me through a prototypical flow of data in a ridesharing company as it would make its way to different constituents.

[00:10:00] TM: So, for that type of scenario, the person who is interacting with the application, so the person who’s getting the ride and the driver will be generating information. So, scheduling the pick up. So, that will factor in things like where the vehicle is located, which incorporates GPS sensor data. All of these have a temporal aspect to it as well or time-oriented. So, those will generally get sent to some sort of inbound processing system, whether it’s a Kafka topic or some other queuing system.

Then from there, you might have a consumer that is pulling off or being able to do real-time analysis of matching a person requesting a ride with the number of drivers who are in the area. So you need to be able to do some fast analytics of what are the distances between those two objects.

Then from there, you might also have a separate consumer that is pulling all of that same data, but putting it into some longer-term archive for being able to do historical trend analysis. So, by having that input buffer, you’re able to have the data serve multiple purposes. Then also for things like Kafka or Pulsar, they have the ability to do some in situ processing to maybe pre-aggregate the data or do some simple analytics to make it easier for downstream systems to then do whatever additional logic they need.

From the consumer who’s doing the real-time matching, they would probably then send another message back down to a push service. They would push a ride request to the driver to know that they need to go and pick up somebody at X-location. Then for the other consumer that’s pulling the data to put into a data lake or a data warehouse, it would just probably push that raw data into whatever system of record for then being able to do different transformations or aggregations for a data scientist who’s trying to determine what are the ways that they can maybe optimize their algorithms to reduce the amount of time it takes to create a match or determine whether they need to have more drivers in a given area to be able to optimize for the customer experience of being able to request a ride and be picked up and dropped off in a more timely fashion.

[00:12:28] JM: It’s useful to have you explain this from a higher level, and this is what you do on your podcast also. I think there’re a lot of people who are getting into data engineering and they may have a limited understanding of these different components. So, getting a bird’s eye view and understanding in more detail how the different areas of the data pipeline works can be useful to these people.

Just to revisit kind of a simple question. Much of the transactional data from this kind of ridesharing application, like the user interacting with the driver, the user summoning a ride, that kind of data is going into a transactional data store. It’s probably going into MongoDB or maybe Postgres or MySQL. We can’t just do all of our operations over that transactional database. We need to get the data from the transactional database and put it into a state that we can manipulate more productively. How do we do that and why do we need to do that?

[00:13:43] TM: So, for the case, say, that we’re loading it into – We’ll go with Postgres for now, because I’m more familiar with it and it has a little bit broader of an ecosystem. But MongoDB has become a pretty heavy contender in this space as well. But Postgres in terms of the default installation of it is optimized for that transactional nature of being able to say with certainty, “I issued a write request. This data is now confirmed. I know that it has been stored safely and I’ll be able to retrieve it at a later data.”

But it’s also generally stored in a row oriented-format. So all of the data for a given record is stored in one sort of file object. Then when you’re trying to do analysis, so for instance we’ll say since we’re talking about ridesharing, there might be a customer I.D. There will be geo-location stamp, so latitude and longitude. There will be a timestamp. There will be maybe some additional metadata of the – It might be related to a foreign key of a city. So, you translate that latitude and longitude into a city object.

But then when you’re doing some analysis later on down the road, maybe all you care about are what are the time distributions across all of these requests? So, you only care about that one column. Whereas in Postgres, you would have to retrieve all of the records and then pull out that column object. So it’s a fairly expensive operation where you’re going to be throwing away most of the data. So, it's extra data over the wire. Its extra data being read from disk, and it's just going to take a lot longer time. Whereas if you're putting it into a column-oriented data store, whether that something like ClickHouse, or SnowflakeDB, or any of the other myriad entrance into that market right now.

If it's column-oriented, you can say, "All I want is this one column," and the database would be able to do a lot of those aggregations in the engine itself and then send back the results to you of what you actually care about, rather than making you do additional processing after you've already returned all of those records. So, it reduces the amount of data sent over the wire. It reduces the amount of time required to perform analysis, and that's massive cost savings both in terms of time and potentially money spent on bandwidth when you're reducing the overall throughput by that level of factor, especially when you're dealing with millions or billions of records.

[00:16:11] JM: There is data all around a large company. There are all kinds of data. How is data permissioned across an organization typically?

[00:16:23] TM: So, that's one that's going to be highly variable depending on the way that the organization is set up. But in terms of a best practice way to do it, you'll usually want to have some sort of single sign-on. So whether that's using Kerberos, or Octa, or some sort of identity system within the company to say, "This is who this person is. This is what their role is," so whether it's a database administrator, or a software developer, or the CEO. So those roles will be associated with different types of permission levels. So, maybe the database administrator will have full access to all of the fields, but an application developer will have any personally identifiable information redacted from their view of the system.

So that central authentication system will be tied into either the engine itself as far as what their permissions are. So, things like Postgres, you can set permissions based on the column level or some other databases will just be a Boolean yes or no of whether or not they have access. Then there are systems such as StrongDM, where they serve as a proxy where they don't necessarily control the specifics of the application permission, but they will audit the entire session so that you can go back later to see if this person was accessing records that they shouldn't have. For a data warehouses system, again, they might have column level permissions.

So, based on what the role is, you would define ahead of time. This role has access to these records in these fields. You'll also want to log when somebody logs in and maybe what queries they're running. So that if you have some sort of compliance audit, you can go back and say yes or no whether or not any of these compliance factors were being violated.

But being able to tie it into a role-based access system makes it much easier for the administrators of the system to be able to maintain their sanity rather than having to every time somebody joins or leaves the company set what are the permissions for this individual. It's more water are their roles. Then you might have layering of roles where somebody is both the database administrator and also acts as a software engineer. Usually, you can figure out what are the set operations for how those roles interact with each other to determine what is the overall calculus for their total set of permissions that are available.

[00:18:49] JM: What role does the data warehouse play in modern data engineering?

[00:18:56] TM: So, the data warehouse is probably this system that has the longest history and probably the most changing role in terms of data engineering in the overall sort of data ecosystem, because data warehouses have existed pretty much since we've had databases and businesses, so going back to the other 80s and early 90s. In those days, it was usually the same type of database engine, but it was maybe put on a bigger server. Then there are also differences in terms of the way that you defined the data model for how everything was recorded. So, you wouldn't just take your application database and then copy it to another server and say, "This is the data warehouse." You'd usually have that transformation step where you convert the data into one of the more popular approaches, is called the Snowflake scheme are the star schema.

But in current times, data warehouses are now leveraging cloud technologies. So, you're able to dump your raw data into something like S3 and then have the data warehouse engine pull that data in and make it accessible. Then you can actually do your transformations using the database engine itself just using SQL. So there are projects such as DBT, or the data build tool, where you actually define all your transformations just as select statements, and then it will create the different views of your data.

But modern data warehouses also are generally column-oriented, as I've been talking about, because it gives you that efficiency improvement of being able to analyze certain subsets of the data across records without having to go record-wise and then column-wise. Also, particularly with the ubiquity of cloud storage, some people have moved to just using a data lake and layering on things like maybe Delta Lake to give it some of the similar semantics of a data warehouse without having the data warehouse engine itself.

One of the main reasons that data warehouses have been going through these transformations is because of the volume of data, where older data warehouse systems just weren't unable to really keep up. So that's where we ended up with things Hadoop and the current movement of data lakes. Also, by forcing the data through a transformation step, there's always the sort of risk that you're going to have loss of information.

So, because of the fact that storage is so much cheaper, people like to be able to store the raw data and then do the transformation after the fact so that if they change their mind about how they want to transform the data or the different views that they want, they still have the original records to then be able to go and reprocess everything. Also by things like more modern file format, such as Parquet, where it itself is column-oriented. It makes it much more feasible to be able to save all of your data in cloud storage or something like the Hadoop file system and then process it using different engines that are built for that that don't necessarily have the data warehouse so you don't have to have as much upfront planning of what is the schema that I want.

So, one of the possible approaches and one that is popular in sort of larger enterprises is the idea of data curation, where you land the raw data in a data lake and then you have a small team that is given access to that to be able to see what types of information they're pulling from that. Then based on their usage patterns, you start to define the necessary transformations that will allow you to evolve stepwise into a more concrete data warehouse format that might power a business intelligence dashboard. So the business intelligence system will have predefined queries that can run efficiently in a data warehouse engine for being able to get high-level views. But then if you do still need to go deeper or perform more detailed analysis, you still have the data lake to go back to using more custom-built tools, or tools that are more suited for doing more of an exploratory approach to the data.

[SPONSOR MESSAGE]

[00:23:08] JM: SpringOne Platform is a conference to learn the latest about building scalable web applications. SpringOne Platform is organized by Pivotal, the company that has contributed to open source technologies, Spring and Cloud Foundry.

This year's conference will take place in Austin, Texas, October 7th through 10th, and you can get $200 off your pass by going to softwareengineeringdaily.com/spring and use promo code S1P200_SED. That code is in the show notes.

Attend SpringOne Platform and work hands-on with modern software. Meet other developers and software leaders and learn how to solve and find out solutions to your toughest scalability problems. Go to softwareengineeringdaily.com/spring and register for SpringOne Platform in Austin, Texas and get $200 off with promo code S1P200_SED. That's S1P200_SED, and that code is in the show notes.

Thank you to SpringOne Platform.

[INTERVIEW CONTINUED]

[00:24:27] JM: When I think about the architectural differences between a data warehouse, database and data lake, I think of a data warehouse as a system that keeps a lot of data in-memory so that it's very easy access. I think of a database as a lot of data on disk, but with some indexes in-memory that make it easier to access those records that are on disk and also the records are probably laid out on disk in a way that's easier to access individual records.

Then in the data lake, I think of a file system with a bunch of files that contain all the data, but not in an easy-to-read quick fashion. Can you tell me your perspective for the different time and space tradeoffs that these different mediums of data storage are making?

[00:25:34] TM: Yeah. So, going in the reverse order. So, database systems I think of as something where you need to be able to have quick real-time read and write access of I need to be able to write records and read records and the specific pattern of read and write depending on the usage, might be a semi-random or it might be bursty. But for the most part, they're going to be fairly equal in terms of read and write volume.

Also going back to what you're saying about the way that the records are laid out. Database systems are definitely optimized for being able to write and retrieve rows at a time so that you can pull the entire information from that. Then for data lakes, it is as, as you said, just a file system usually with metadata management system accompanying it for being able to track what the data is, where it's located, maybe some of the semantics around the data, such as who owns it, when it was produced. So that you have some way of keeping track of what data is where, but it's much more just free-form access of being able to pull different records from different places and experiment with how you want to use it.

It's not as prescriptive in terms of how the records are related to each other or even if they're related to each other. So, it's generally going to be slower in terms of the read speed versus what you have in a database. Databases are also generally going to be record at a time transactions, or may be, at the most, on the order of a handful of records at a time within a given transaction. Whereas dealing with data lakes, you're probably going to be writing data into it in batches of dozens or hundreds or thousands of records. So, on the order of megabytes or gigabytes versus kilobytes or megabytes in a database.

Then in a data warehouse, you're sort of going back down to a smaller scale of data than the data lake, but still much larger than the database where the data is written in a way where it's easy to retrieve the data in sort of a contiguous fashion. But rather than it being record at a time, it's probably going to be column at a time. You're probably going to be writing data into it in batches, rather than individual records like you would with a database. You're also going to be retrieving aggregates of data at a time versus single or maybe dozens of records at a time like you would from a transactional database, or I should say relational database.

Also, data warehouses are much more structured in the way that the data is represented versus a data lake so that you are able to create indexes on them to optimize for the read patterns, particularly for things like business intelligence dashboards or standardize reports. So, data warehouses are also different in terms of the – I guess all three of those are different in terms of the consumer of the data.

So, a relational database or document database is usually going to be interfaced with some sort of application for reading and writing the data. It's usually not going to be a human doing all that interaction. Whereas a data lake, it's probably going to be much more of an exploratory use case where you have some sort of machine system bulk loading data into it, but then you have a human driving the other side of pulling data out for some sort of analysis until you get to a point where you have some sort of automated flow of maybe performing machine learning on it or doing some extraction of the data from the data lake into a data warehouse or into some other secondary system.

Then from the data warehouse, it's probably going to be some sort of a business intelligence dashboard or some human doing SQL queries against it to do some sort of interactive analysis of the data in the system. So, differences in terms of the volume, in terms of the speed of access and the sort of patterns of read and write, and then also the consumer of the data. So, data warehouses are probably going to be more business end-users. So, CEOs, CIOs, CTOs, people who maybe working in marketing or sales to be able to pull reports, whereas a data lake is going to be more data engineers and data scientists working on it and an application database is going to be more software engineers and software systems interacting with it.

[00:30:06] JM: How does data make it out of the data lake and into a data warehouse and what data is getting selected to put into that data warehouse from the data lake?

[00:30:20] TM: So, going from the data lake to the data warehouse, it’s usually going to be managed by some sort of ETL tool, although with modern data warehouses, a lot of them are adding the capability of ingesting data directly from the engines. So, things like ClickHouse and MemSQL both have capabilities built in for being able to pull data from some sort of cloud storage or data lake and then load it directly into their system.

As far as the selection criteria, it's usually going through a point where you land the data in a data lake and then you have your data scientists or data engineers who are interacting with that trying to build reports from it and then figuring out what works and what's useful. Then after they have come to that conclusion, they will then codify those transformations for determining how that schema is represented and what gets loaded into the data warehouse. They'll also figure out the frequency of data loading. So whether it's daily batches, weekly batches, or if they're going to use some sort of a streaming system for being able to write the records into the data lake, but then also perform a transformation on them and write them directly into the data warehouse at the same time.

[00:31:37] JM: I have heard that data warehouses can be expensive. Why is that?

[00:31:43] TM: Usually because data warehouses are trying to optimize both for large data volumes and also for speed of access. So, for a data lake, you can usually get away with using inexpensive object storage and accept certain levels of latency. Whereas for a data warehouse, you're trying to achieve similar volumes of data. So you might have gigabytes or petabytes of data stored directly in the data warehouse itself, but then you also want to have it formatted and served in such a way that you can do interactive analysis of it.

So, maybe things like BigQuery or snowflake where you want to be able to process petabytes of data in a single query and get a response back in the order of seconds to minutes. Whereas with a data lake, you might be working more with like a MapReduce style approach or being able to accept responses on the order of minutes to hours.

So, because of the fact that you're trying optimize for both size and speed, you're going to be working with a higher level of compute capacity as well as faster, more near line data storage and also higher bandwidth throughput. So it it's just because of the fact that you're trying to optimize for speed and storage, but not quite on the same scale as the speed of a database or the storage of the data lake. So it's sort of in between those two in terms of the system requirements. So, it's just inherently inexpensive problem.

[00:33:19] JM: Distributed queues like Kafka have played an increasing role in data engineering. How are people using distributed queues?

[00:33:29] TM: Pretty much anywhere that you can imagine. So, at first, it was just a solution for the problem of reliability of queuing systems. So, we've had things like RabbitMQ and ActiveMQ for quite a while of being able to publish data and then have multiple consumers. But they all have some tradeoffs or some issues as far as the reliability of the data and the durability of it. So, Kafka was one of the first movers on that front as far as having a persistent log structured queue, where everything is stored potentially forever, but in practice, on the order of days or months and then being able to go back in history and reprocess all of those messages, but also have different consumers consuming at different rates.

So, then they added on other capabilities such as being able to have functions that execute on the records as they flow through to maybe generate alerts or perform some in-memory aggregations within a certain window. So maybe windowing the last 50 messages and figuring out what's the rolling average. Just the fact the you have this historical record, this durability guarantee and the ability to have consumers being able to use the data for different purposes and at different rates means that you're really only limited by your imagination. As far as how you want to use the data and where you want it to go.

In practice, it's often used for being able to feed multiple downstream systems from a single data source. So, in the event of clickstream analytics, you might have your data going into a transactional system for being able to interact with it in sort of the near range time window. You also have those same records going into your data lake for long-term storage. Then you also have another consumer that's doing some lightweight transformations to write it into a data warehouse. So you have one data record being fed into multiple downstream systems without having to do that fan out from the source system itself. You can just have one target for a data creator to write to and then it doesn't have to care about what are all the downstream systems that are going to want to use this down the road.

[00:35:53] JM: How do you define the term data stream?

[00:35:57] TM: So, a stream of data, the way I think about it. It is either – Generally, individual records being created in what’s termed as sort of an unbounded stream. So, abounded stream would mean I have five records that I create. That's all there is. An unbounded stream means I might have five records. I might have 5,000. There is no way for me to know ahead of time how many records there are going to be. So I need to be able to just process them as they arrive.

A stream of data could also be what we like to think of in terms of streaming for music or movies, where you have just a flow of bits that are encoded to represent music or your favorite Netflix show. But in the data engineering context, it's generally not quite that same terminology. For data streams, we're usually talking about streams of individual records or events that have some sort of relation to each other in terms of the semantics of what they're producing, but not necessarily in terms of their contents.

So, a stream of data coming from an IoT sensor, you might have 5 million sensors deployed. All of them are writing into the same data stream. So they're all the same and that they're coming from a sensor, but not necessarily the same sensor or the same location or the same timeframe. Whereas a movie stream, it's all for the same movie, all contiguous. Each packet is representing the subsequent frame in the movie.

[00:37:30] JM: I think one thing we can be confusing is when you think of a stream. You think of this moving body of water where if you put your hand in, the stream just passes you by. If you don't, if you weren't to grab up the water and pull it into your hands, the water would just make its way past you. You wouldn't be able to reclaim it.

Well, the way that a stream works in data is that it is just this append-only thing. So it's not like if you don't catch the stream right now, it's going to go disappear. It's just added on. Its appended to continuously over the lifetime of that stream.

[00:38:13] TM: Yeah. I think a more useful way to think about it is, as you're saying, in a stream like you would see out in nature, it's constrained by its banks, but not by anything else, and it's – There is a start and an end, but when you're experiencing it wherever you happen to come across it, you don't know where they are. A more useful analogy might be to think about the datastream as it's flowing through a pipe in your plumbing system, where it's all still water, it's all still flowing, but there are constraints in terms of where it can go and how much it can contain.

So, in these big data distributed system streaming engines, such as Kafka or Pulsar, on some level you think about it as being essentially infinite in terms of capacity, but in reality it's not. There is some capacity limit as far as what can be contained. If you don't have some sort of consumer pulling data out of it or some sort of lifecycle management policy for how long a record stays, you are going to end up filling it up. Then you might experience what's called back pressure from the source system where the source wants to write data into it, but the streaming engine might either respond with a negative acknowledgment to say, “I can't accept it,” or the data might just get dropped on the floor. So, you want to be able to process the stream as it flows through or you are going to start having – You might end up with a flooded basement.

[00:39:35] JM: Have you looked at the differences between Kafka and Pulsar, which are I think the two popular open source distributed queuing systems?

[00:39:46] TM: Yeah. So, Pulsar has the advantage of being the second mover in the space. Kafka was the one that sort of started this whole trend. So, Kafka came out of LinkedIn and it was focused on being able to reliably store events for a long period of time, and it has a notion of topics. So the way that you scale a Kafka system from a logical perspective is you add more topics to it. So, different topics usually correspond to different types of events that will be contained within them.

So, the system itself will scale, but the compute and the storage are sort of interlinked in the way that Kafka is built. So, you deploy a server. It needs to have sufficient storage and compute processing capability for being able to handle the different topics.

Pulsar came out of Yahoo and it is a little bit more flexible. So, Kafka is just append-only log record. Whereas Pulsar can be treated in that same fashion, but it also has the capability of being able to represent things like AMQP and used more. It’s just a traditional pub/sub message like you would do with Rabbit MQ. It also has – For the storage layer, it uses something called Bookkeeper, which is another Apache project.

So, that means that you can scale the compute and the storage independently and it also has built-in automatic life cycling of the data. So you can set a lifecycle policy so that any records that have been in storage for longer than five days automatically get written to S3 or Google

Cloud Storage. So, it means that the amount of data that it can store from the end-user perspective is larger, but the system itself doesn't actually have to be able to have all of it directly accessible. Then you still use the same API for interacting with it. You just might have to accept certain levels of latency for older data that's been tiered out to other object stores.

Pulsar also, because of the fact that they're a second mover and Kafka was already such a significant force in this space, they added APIs that make it compatible with Kafka so that, in theory, you can actually just drop it into a system where you're already using Kafka and not have to change your application code at all. So, Pulsar has a little bit of benefit in terms of flexibility and in terms of the deployment as well as in terms of the usage patterns.

Kafka, however, has a much larger ecosystem built up around it of systems that are already built to operate with it. So, they're both very good systems from what I've been able to tell. They're both written in Java. They're both Apache project. So, they're pretty well supported and well adopted. But definitely worth looking at the two of them if you're in a new system and you're trying to figure out which one to use in which one fits your use cases better. Pulsar, I know also has a Pulsar functions capability for being able to execute arbitrary code on individual records. Kafka also has a similar capability.

[SPONSOR MESSAGE]

[00:43:03] JM: As a software engineer, chances are you've crossed paths with MongoDB at some point. Whether you're building an app for millions of users or just figuring out a side business. As the most popular non-relational database, MongoDB is intuitive and incredibly easy for development teams to use.

Now, with MongoDB Atlas, you can take advantage of MongoDBs flexible document data model as a fully automated cloud service. MongoDB Atlas handles all of the costly database operations and administration tasks that you'd rather not spend time on, like security, and high-availability, and data recovery, and monitoring, and elastic scaling.

Try MongoDB Atlas today for free by going to mongodb.com/se to learn more. Go to mongodb.com/se and you can learn more about MongoDB Atlas as well as support Software

Engineering Daily by checking out the new MongoDB Atlas serverless solution for MongoDB. That's mongodb.com/se.

Thank you to MongoDB for being a sponsor.

[INTERVIEW CONTINUED]

[00:44:25] JM: Eearlier you mentioned the evolution of the data Lake and the data warehouse in terms of Databricks Delta, or I guess maybe it's called Spark Delta or something. Anyway, the Delta Project. Which I know came out of Databricks. Can you describe what you're seeing there and evolution between the data lake and in the data warehouse and how Delta Lake epitomizes that, or just describe that project in more detail.

[00:44:57] TM: Sure. I actually had the pleasure of interviewing one of the architects of that project on my podcast a little while ago.

[00:45:02] JM: I heard that. It's a great episode.

[00:45:04] TM: Thank you. So, Delta Lake is one of those projects that sort of blurs the line between data lake and data warehouse. So, the way that it functions, it is all built on top of Spark as the engine that codifies it. It standardizes on the Parquet file format. It uses whatever sort of object storage. So whether it's the Hadoop file system, S3, Google Cloud Storage. It adds transactional semantics on top of the data so that you can have asset guarantees for the way that your writing your records out.

Then it also has built-in capabilities for setting expectations of what the data is going to be so that you can have quality controls of the data as it flows into your system. Then one possible design pattern is that you can land all of your raw data into your data Lake and be able to make use of it, but then you can progress it through different levels of sort of curation. So, you land the raw data in and then you define some transformations to write it out into a separate set of table spaces that are more structured and more akin to a data warehouse and then graduate to different levels until it's into a sort of well-defined schematized fashion the same way as what you would interact with in a data warehouse, but it's all still in just your object storage of choice and your interface is still the Spark APIs.

So, you can use SparkSQL to query it. You can use all the other capabilities of Spark for being able to do machine learning on the data. It all still resides in your data lake, but you're able to have the same types of quality guarantees and transactional semantics is what you would have been a data warehouse. So, depending on the business that you're in, you might actually not need to have a data warehouse at all because of the fact that you getting some those guarantees from the Delta Lake project.

[00:47:02] JM: Tell me about the typical data engineering problems that are being encountered by enterprises. So, I know you work as a consultant and you also talked to a lot of different companies. Just give me perspective for what the average company is going through in terms of data engineering.

[00:47:24] TM: So, I think one of the biggest problems right now that companies are going through is the issues of data cataloging and data discovery and the issues of data quality and what's called master data management. So, particularly for large enterprises, you'll have a lot of different systems that are going to be producing data, different systems that are consuming data at a higher level. So the CEO of the business, they want to know what data do I have. That's a big problem of – It's a hard problem to answer. So, that's where you're seeing a lot of metadata management systems and data cataloging systems becoming more prevalent in the last year or two. So, that's for being able to hook that into the overall lifecycle of all of your data systems so that you can say, "Do I have any health information?" and then you will say, "Okay. It's in this application database over here, or it's in this data lake over here. This is the schema for it. This is the last time there was a record written to it. This is the owner or the person on record who I need to contact if I have any issues with it or if I want to find out more."

So, just being able to know what data there is, what ways is it being used? What jobs are producing it and consuming it? Being able to track the overall lineage of the data from when it was first created all the way to all the end-users of the data system. So there's a project called Marquez out of WeWork. There's a project called Amundsen out of Lyft that are both trying to solve similar problems in that space of discovery and access and being able to track the lineage of the data.

Then the other issue that is a corollary to that is master data management, where you have five different systems that interact with your customers. You have your CRM. You have your application databases where customers are purchasing from. You have information coming from social media channels. You have website analytics of people who haven't signed in, but they're interacting with your different web properties and you want to be able to say, "What are the interaction patterns of one individual across all of these channels?" So you need to be able to have some way of correlating all of those different systems and the records within them to say with some level of confidence that these are all related to this customer or these are all related to this product and then being able to perform analyses across all of those. Then that master data management system ties into all the other downstream analytical systems for determining what are the semantics around the data. How do I define what the user is? How do I define what a customer is? What a product is, and which products are all pointing to the same – Or which product IDs are all pointing to the same actual product in my system?

[00:50:16] JM: From the data engineering problems that you see, are they largely technology problems or are they more rooted in process problems, or like problems of human dynamics and the roles not being properly defined and those kinds of things?

[00:50:35] TM: So, technology is a piece of it, but in my experience, just the human factors are generally what lead to a greater deal of complexity in any technological system. So, taking the master data management problem as a case in point, if it was just a matter of technology, then you would have somebody define these are the ways that we track all of these different attributes across all of these different systems. Then you don't even have to have some sort of sophisticated matching technology downstream to try and recombine things, because that would already be solved.

But because you have different people and different systems with different ideas of how things need to be represented all creating these different data records, you then have this messy problem of how do I join them all back together to be able to get a cohesive view of how a customer is interacting with all of my different products. Similarly, for click stream analytics for instance, you might have a schema that's supposed to come in from your system, but then different users will be having different ad-blocking technologies or they might have different plug-ins that will corrupt the data as it's coming in. So, there are a lot of different human elements that come into it. But then there are also technological aspects of the network is unreliable, so packets might get dropped.

So, I don't know with any certainty that I'm getting all of the data from the source system. So, for instance, for IoT, this sensor might be sending data that I'm not retrieving and I don't know it unless I have some way of sequencing the transmission IDs. So I know, "Okay. This has ID five, and then the next packet I get is ID15. So I know I lost 10 packets, but I don't know what's in them and I have no way of recovering them." So, it's a combination of technical and human aspects, but the human aspects are generally the ones that are more challenging to overcome.

[00:52:28] JM: Can I get a full date engineering stack from a cloud provider or do I need to go outside of the cloud provider and the symbol set of tools from open source software?

[00:52:39] TM: So, Amazon is definitely doing its best to make sure that you can get everything under the sun from them, but there are number of cloud providers that are doing a really good job of providing a lot of the primitives for being able to build your end-to-end solution using just all of their managed services. You might still have to do a little bit of glue to tie some of them together, but the state-of-the-art right now for cloud providers is definitely progressing where, depending on your needs, you could just use everything off-the-shelf from your cloud provider of choice and not have to pull in anything open source. It really depends on what are your needs. At what scale are you operating and how comfortable are you just using a black box versus having an open source product that you can tinker with and delve into on your own and understand why things are breaking at the end of the day.

So I'd say for small to medium scale businesses or small to medium scale projects, it should definitely be possible to just use cloud offerings, but when you get into sort of uncharted territories or larger scale analytics, you're probably going to want to start bringing in either different open source options or building in your own custom capabilities that fit your specific use case.

[00:53:56] JM: What do you think of the term NewSQL?

[00:54:01] TM: I think that it is just more sort of industry jargon, not wanting to admit that everybody was wrong when they said that SQL was dead. But in terms of what it's actually intending to represent –So to give a bit of history. So, there was the era of SQL, which is relational database systems, so Postgres, MySQL, Oracle, Microsoft SQL, all those different systems and they scaled to a certain point. But then when we started hitting the internet era where we were generating petabytes of information, those systems weren't able to scale horizontally enough and be able to provide the performance that we needed. So we ended up with what people were dubbing NoSQL systems, where they didn't have those transactional semantics and ACID guarantees. They were built more for horizontal scale and there wasn't necessarily any consistency guarantees as far as a record being written in one place and represented everywhere. You usually ended up with eventual consistency.

They also didn't always have a predefined schema. So, they were called schemaless, but it just means that it was schema on read instead of schema on write. Then there were a few years of that. People started to realize that that had its own set of problems. I mean, they're great technologies for certain use cases, but they don't solve everything.

So, now there is a move back to using SQL as the API interface and then also moving back towards relational semantics and transactional semantics, but building in horizontal scalability. So, things like CockroachDB, Google Cloud, Spanner. FaunaDB is sort of in that camp, but they don't actually use SQL as their language. Essentially, it's databases that are moving back to using SQL as the interface. They're going back to using relational semantics, but they're built in a way to be able to scale horizontally so you can get much higher availability, and distribution, and scalability. But you're still using the familiar SQL interface that you're used to. So, that's usually what I think of when I hear NewSQL. So it's a return back to relational semantics, but with horizontal scaling built-in.

[00:56:12] JM: How has Kubernetes changed the world of data engineering?

[00:56:15] TM: So, Kubernetes is an interesting aspect of the data engineering landscape right now. I don't think it was ever really intended originally to be a solution for the data problem. It was more for application deployment and management. But because of the fact that it is a distributed systems framework, it means that a lot of people are trying to use their other distributed systems within this for deployment and maintenance, because deploying and maintaining distributed systems is hard and Kubernetes has made that simpler. I'm not going to say easy, because it's still not easy, but it's easier to reason about, and there are also a lot of managed offerings so that you can run a lot of your application stack within Kubernetes and use a similar set of APIs and semantics for how your entire system is managed.

But I'm definitely seeing a lot of activity of different database vendors and open source database systems as well as tools like Spark and Kafka adding what are called Kubernetes operators to make it easier to deploy and manage these data systems within a Kubernetes environment.

So, one of the challenges that was faced originally was the data persistence problem, where, originally, Kubernetes was intended for stateless workloads where you have a web server, and if it needs to be destroyed and rebuilt or moved to a different physical server, it didn't really matter because it didn't have any of its own data that it needed to care about, which obviously isn't the case for databases.

So, now, Kubernetes has the concept of stateful sets, where you're able to say with some guarantee that this service or this application or process needs to be located with this actual data storage. So, it's just a set of API primitives. In terms of how it's actually implemented is up to the distribution of Kubernetes or the specific installation, but it means that when you deploy a database application, it's going to be located with its data. So, you still get some of the guarantees of ease of deployment with the associated guarantees of this data is where this application is going to be deployed with its data. So, that's made it much more possible for bringing some of these data-oriented workloads into Kubernetes itself.

One of the more interesting entrance on the sort of Kubernetes and data engineering space is Pachyderm, which is leveraging containers and cloud storage for building an entire data science pipeline within Kubernetes itself focused on Docker containers and cloud storage for being able to provide versioned datasets.

[00:59:03] JM: It's 2019. You and I did show a little more than a year ago. What have you learned about date engineering in the last year?

[00:59:10] TM: That is a good question. So, I've gained a lot more perspective on the common themes in the data space as far as what are the challenges that are somewhat universal across organizations and across projects. I've learned a lot more about some of the differentiating factors for projects that at face value seem like they're doing the same thing.

It's confirmed a lot of my initial ideas around what I like to call mechanical sympathy of making sure that whenever you're building an application or a system, that you’re keeping in mind the physical limitations that are inherent in the universe and making sure that you are optimizing your architecture and design for what’s actually possible with these different systems. I’ve also learned a lot more about some of the deeper details about database systems and some of the concerns that go into those. Just in general, gained a lot more perspective and broad understanding of the entire landscape.

[01:00:19] JM: Tobias Macey, I’m a fan of your podcast. Thanks for coming back on the show.

[01:00:23] TM: Thank you for having me again.

[END OF INTERVIEW]

[01:00:28] JM: GoCD is a continuous delivery tool from ThoughtWorks. If you have heard about continuous delivery, but you don't know what it looks like in action, try the GoCD Test Drive at gocd.org/sedaily. GoCD's Test Drive will set up example pipelines for you to see how GoCD manages your continuous delivery workflows. Visualize your deployment pipelines and understand which tests are passing in which tests are failing. Continuous delivery helps you release your software faster and more reliably. Check out GoCD by going to gocd.org/sedaily and try out GoCD to get continuous delivery for your next project.

[END]