EPISODE 1060

[INTRODUCTION]

[00:00:00] JM: Data engineering typically involves both a data lake and a data warehouse. A data warehouse is a system for performing fast queries on large amounts of data. A data lake is a system for storing high-volumes of data in a format that is slow to access. A typical workflow for a data engineer is to pull datasets from this slow data lake storage into the data warehouse for faster querying.

Apache Spark is a system for fast processing of data across distributed datasets. Spark is not thought of as a data warehouse technology but it can be used to fulfill some of the responsibilities. Delta is an open source system for a storage layer on top of a data lake. Delta integrates closely with Spark, creating a system that Databricks refers to as a data lakehouse.

Michael Armburst is an engineer with Databricks and he joins the show to talk about his experience building the company as well as his broader perspective on data engineering and his work on Delta, the storage system built for the Spark ecosystem.

I also want to mention, I've started doing some investing. If you are an entrepreneur with a great idea particularly one around engineering, send me an email, jeff@softwareengineeringdaily.com.

[SPONSOR MESSAGE]

[00:01:23] JM: This episode of Software Engineering Daily is sponsored by Datadog. Datadog is a cloud monitoring platform built by engineers for engineers enabling full stack observability for modern applications. Datadog integrates seamlessly to gather metrics and events from more than 400 technologies including cloud providers, databases and webservers. Easily identify slow running queries, error rates, bottlenecks and more fast with built-in dashboards, algorithmic alerts and end-to-end request tracing and log management from Datadog.

[INTERVIEW]

[00:02:28] JM: Michael Armburst, welcome to the show.

[00:02:30] MA: Yeah, thanks for having me.

[00:02:32] JM: You joined Databricks 6-1/2 years ago and around that time Spark was becoming popular because it could do large scale processing without checkpointing the intermediate datasets to disk. I believe this was the main issue with people using Hadoop around that time. Tell me about the early engineering problems that you worked on at Databricks.

[00:02:55] MA: Yeah. I knew the Spark guys back when they were working on it at UC Berkley, but I didn't actually become a Spark user myself until joining Databricks. When I joined, really my mission was SparkSQL. I think you were right in identifying that checkpointing to disk was very expensive, but to me the other kind of magic trick that Spark always had was this kind of high-level, really simply APIs that made it elegant to express what you were trying to do.

But still, that was kind of still functional programming. You're telling it what needs to be done and also kind of how it needs to be done. The magic of SparkSQL is we wanted to take all of that niceness of Spark, all of the performance and the power and we really wanted to make the APIs even simpler to use. We wanted to make them declarative so that you say what can be computed rather than how it's going to be computed, and then let the system figure out what optimizations to create.

[00:03:44] JM: As Spark was becoming popular, there were lots of streaming systems. There were Storm, Flink, Heron, Apache Beam. You worked on structured streaming eventually at Databricks. What are the key design decisions in building a streaming system?

[00:04:03] MA: Yeah. I think the main motivation here, kind of our guiding principle, was we wanted streaming and batch to be unified. We wanted any streaming query to guarantee that it would return the exact same answer as in batch query, and this was because of a lot of complexity we saw in this kind of earlier streaming paradigm known as the land architecture, where you would have one system that was running really fast, kind of fast and lose, maybe dropping some packet somewhere, but it was giving you this like very low latency streaming. Then you'd have another system that was this big, slow Hadoop monster that was maybe computing once a day but would give you the accurate answer.

Our goal with structured streaming was we really wanted to bring these two worlds together so that you could write one program and then just choose in a kind of cost latency tradeoff what was the right balance for your application.

[00:04:53] JM: Structured streaming was an advance or an alternative to the original streaming model that Spark had first developed, right? The micro-batch streaming model? Can you describe how structured streaming compares to the micro-batch model?

[00:05:08] MA: Yeah. Well, okay. There're a couple of things I want to unpack here. First I would say really the biggest difference between DStream, the kind of first version of streaming that existed inside of Spark and structured streaming is again this difference between imperative and declarative APIs. DStreams, you are still selling the system how you want something to be computed. If you're computing in average, you're actually calculating the sum and the count and dividing them yourself.

Where is in structured streaming, we were building on top of SparkSQL. We wanted to start with the same high-level declarative APIs where you say what you wanted to accomplish rather than how and then let the system figure out the right way to do it. But there were actually a bunch of other things we wanted to change here. One was we wanted to not be tied to a particular computation model. So that was kind of what I mentioned before where we wanted to have this high-level semantic that you get the same answer with a streaming query as a batch query, but we didn't want to be tied necessarily to micro-batches.

Now, kind of one thing that's important to understand here is structured streaming actually supports multiple different computation modes. If you have something where you're looking for latency on the order of tens of seconds to minutes, micro-batches are still actually a really efficient way to get it done. So structured streaming does support micro-batch computing still.

The reason that's really efficient is because batching is kind of one of the canonical engineering tricks to get good throughput. This is why structured streaming has the highest throughput of any streaming engine that we've ever compared against.

Now, we didn't want to be tied to that, because in some cases you don't care about throughput as much as you care about latency. Really, in the query language itself, when you're writing a query, you say nothing about what computation model you use. It's only when you actually start the query that you pick. What executions strategy to use, and we can then also run in continuous processing mode where rather than give up the resources when you're done processing any individual set of records, you hold on to them and you're just continually pulling, processing all the time and there you actually can get the kind of millisecond latency that you see in other streaming systems that don't use micro-batches. Really, I think it's really about giving users a choice where they can decide cost latency. What makes sense for their application?

[00:07:17] JM: I see. In the lowest latency form of structured streaming implementation, you're going to have your structured streaming system continually pulling whatever the egress point is for the data that you're sucking in and that's going to be expensive computationally from all the pulling resources?

[00:07:40] MA: Yeah. I think it's not really about computational expensiveness. It's really about kind of resource sharing. You are holding on to that core the whole time. Spark is not scheduling any other work there, and in most streaming clusters, you're not doing just one thing. You have a whole bunch of different applications that you're trying to run.

The way I like to think about it is continuous processing, you're holding on to that core continuously. Nothing else can take advantage of that even if there's no data coming in. That's kind of the price that you're paying for low-latency. Whereas in micro-batch, you are multiplexing

the resources of the cluster on a really find granularity. Every 10 milliseconds we're deciding what should be running on every core, and that way you can have a whole bunch of things running. When there's a lot of data coming in, one stream might take more of the resources. When there's not data coming in, then maybe we won't be taking any resources for a little while. Does that make sense?

[00:08:29] JM: It does. Yeah. That model of the continuous pulling and just reserving one of those CPUs for the continuous pulling, how does that model of low-latency ingest compare to the streaming model that Flink has?

[00:08:50] MA: Yeah. As far as I understand, it's actually very similar. Both systems kind of have a relatively static kind of configuration of what executors are participating in each stream. If you want to change it, there is kind of a relatively expensive bring everything down, start them back up. But as long as it's running, everybody is continually pulling, getting super low-latency. I think you get kind of comparable latency semantics from either system.

[00:09:15] JM: The fact that in structured streaming, as you just explained, you can choose whether you're going to micro-batch or continually pull in new data points. That kind of sounds like the Apache Beam perspective – So if I understand the Apache Beam perspective correctly, it's that there are these different circumstances that call for different stream processing engines partially because of the kinds of tradeoffs that you explained. But if Spark already gives you the options, that spectrum of options, within the structured streaming, it seems like you can have the potential to explore a similar spectrum of options that Apache Beam would give you. Do you know much about Apache Beam or can you draw comparisons there?

[00:10:05] MA: Yeah. I've always kind of thought Apache Beam in terms of the APIs that it provides. It's, again, much closer to kind of the RDD, the original styled Spark. You are telling the system how computation needs to be done, and this is why I really – I think SparkSQL, having these higher level languages, it opens up the possibility of using the system to many other people.

You're right that Beam kind of always had this idea of targeting a whole bunch of different runtimes, but to me that's a really difficult problem to solve. You often end up kind of with like

lowest common denominator semantics, and I know at least like one of the things I've heard is use Beam, you can run on Spark. I have never had anybody have a very successful outcome actually using it, because the Spark Runner isn't nearly as good as the runner that they have inside of Google.

I think with Spark, we're solving a much easier problem. Even though continuous mode and micro-batch mode have slightly different execution semantics, you're still using the same underlying code to do the processing. There's much more of a kind of unified substrate for these two things to run on. I think it's just easier for us as engineers to provide a rich experience.

One thing I will say though just to be fully upfront is continuous mode, we're still doing work on it. It is missing some of the stateful operations that you can do in micro-batch processing in structured streaming today, although that's certainly not anything fundamental. It just hasn't been where the community had been spending their time.

[00:11:30] JM: Got it. Now, just going a little bit deeper on Apache Beam for a moment, what's your understanding of what Google was strategically trying to accomplish with the whole Beam thing? Like the main strategic point I saw was Apache Beam was kind of an onboarding API for getting users started on the Google like manage dataflow thing, but this was kind of like a rare example of like an open source, or an open core Google strategy that I always had trouble understanding exactly what they were trying to do or what their end goal was. Do you have a perspective on what they were trying to do strategically or what they are trying to do strategically?

[00:12:18] MA: Yeah. I mean, this is pure speculation, but I think you really hit on it when you said openness. I think if we look at anything that has really shaped the big data ecosystem in the past decade, it's been the rise of open source as the way to do processing of large amounts of data. I think there's a really good reason for that. I think when you're an organization and data is one of your most valuable assets, it turns out the people who write the code to process that are the most expensive part of that equation.

You do not want those computations to be locked in to one particular vendor. You need to be able to move them if Google decides, "Hey, this week, we don't care about data processing

anymore and shut it down." You need to have some story. I think my guess is what they were trying to do with Beam is they saw kind of the writing on the wall. People are not going to work with data processing systems that are not open. This was their attempt to create open APIs that could still be powered by all the cool Google tech underneath the covers.

The problem to me though is there never really was a first-class open source Beam runner, whereas if you look at things like Spark and Flink, they started in open source. There is a great kind of first-class execution engine out there that is totally open, and I think that's why at least in my experience and the people I work with, I think there's a lot more uptake of these other APIs.

[00:13:40] JM: Yeah. Well, I can almost imagine the conversations in the halls of Google where they're like, "Yeah, we really want to make this dataflow thing that we have internally that everybody loves more accessible, but all the people that are wanting to do this parallel processing out in the rest of the industry want to use open source stuff. What's the middle ground?" and they end up with Beam and maybe it seemed like a really good idea at first blush. Maybe it has a lot of users, but I don't know if it does. But it's kind of interesting project to contrast with Kubernetes where they said like, "Look, let's just go whole-hog and have a clean room reimplementation with – We've got these papers that have written that are kind of reference implementations, but let's go with a completely new project," which has been more successful at least from what I can tell.

[00:14:32] MA: Yeah, I think you're right. That's a really interesting contrast. I haven't thought about it exactly like that, but Databricks is powered by Kubernetes under the covers and we don't yet run on Google. I think it's exactly what you're saying though. Kubernetes, there is a first-class reference implementation that is world-class. When it came out, it was better than anything else and it was all out there, and I think that's a much better way to do open source.

[SPONSOR MESSAGE]

[00:15:03] JM: JFrog Container Registry is a comprehensive registry that supports Docker containers and Helm chart repositories for your Kubernetes deployments. It supports not only local storage for your artifacts, but also proxying remote registries and repositories and virtual repositories to simplify configuration.

Use any number of Docker registries over the same backend providing quality gates and promotion between those different environments. Use JFrog Container Registry to search the custom metadata of the repositories. You can find out more about JFrog Container Registry by visiting softwareengineeringdaily.com/jfrog. That's softwareengineering.com/jfrog.

[INTERVIEW CONTINUED]

[00:15:57] JM: Okay. Well, I want to get into talking about the stuff that you've worked on more recently, data infrastructure, particularly data lake infrastructure or data lakehouse infrastructure, depending on what terminology you want to use.

Over the period of time that you've been at Databricks, there's been significant adaption of Spark, and the spark jobs are often pulling data out of the data lake such as HDFS or S3. Describe the problems that you see with the data lake abstraction.

[00:16:30] MA: Yeah. That's a great question. Maybe I actually kind of first want to start with what I think is right about the data lake abstraction, because I think it's important to understand what properties you want to preserve before you go changing it. To me, the reason data lakes kind of first started to take off is there's a couple of things. First, organizations have a wide variety of types of data and it has a lot of different kind of value, right? There are some data that's super high value. It's well-structured. It's like coming from your ERP system, and people spent a whole bunch of time creating it. You definitely want that.

There's also a lot of low value data. It's just telemetry and metrics coming out of systems. It's generated by machines. There's audio and video, and the magic of the data lake was it allowed you to just collect all of this in one place without spending a bunch of time doing ETL. You might say, "Well, why do I want to collect everything," and it's because you don't know what's valuable until later. If you didn't collect it, then it's gone forever.

What the data lake got right was it allows you to take all of these data and just dump it in one place and come back to it later, and it did that with a couple of other properties. It was super cheap. HDFS started running on commodity hardware, but actually things like S3 and ADLS I

think took this to the next step, because now not only are you running on commodity hardware, but you're getting the economies of scale that Microsoft and Amazon can produce.

Again, it's super elastic. In HDFS, you just throw more machines at it. An Azure or AWS, you just use it. It just gets bigger, it just kind of works magically. I think those are the properties that we really liked about the data lake. However, a data lake is definitely not a database. You are kind of taken back 20, 30 years and you have to re-implement a whole bunch of kind of complicated systems solutions that you didn't have to do when you were using a database. In any distributed system, one of the like biggest challenges is dealing with partial failures. I run a Spark job and it gets halfway through and it dies. What I do?

On a data lake, you have no protection. You've written out a whole bunch of partial data. You've maybe even like overwritten stuff that was there before. There's no easy way to roll back. Again, a data lake is just a file system. You have no enforcement of schema. It's possible for anybody to come in and just drop data in that directory that doesn't match the format that you're carrying about. There's also no control for concurrency. If multiple people are trying to modify the same dataset at the same time, there's really no guarantee as to who's going to come first or what's going to happen.

When we started creating Delta, we kind of sat down and we said, "How do we make this thing look more like a database?" The kind of key answer to that, I think, I think one of the coolest things that database has got right is this idea of ACID transactions. When you do any operation, it's atomic. You know kind of for sure whether the entire thing happens or nothing at all. You kind of completely get rid of that partial failure problem. It's consistent and it's isolated. Even if multiple people are modifying it at the same time, it actually acts as the one person who's going at a time, which makes it very easy to reason about correctness. Then durability, we just wanted to make sure we didn't get rid of that. These storage systems were already pretty good at that. We want to make sure that we maintain those guarantees.

Basically, our goal was how do you bring ACID transactions into this system while still maintaining those really nice properties of elasticity, and low cost, and being able to collect everything without doing a bunch of expensive ETL.

[00:19:51] JM: Right. Now, I just like to get your perspective on how data typically makes it into a data lake. What is the format it's in and what's the infrastructure? What are the steps in the infrastructure that are happening before a piece of data actually gets put into the data lake?

[00:20:10] MA: Yeah. I mean, that's a great question. Here is where I think spark really shines. I think – I've always looked at Spark as kind of the skinny waist of the big data ecosystem. I think one of the things we got right back in the like Spark 1.2 days was building this data source API so that people could build connectors into Spark, and so you could read from everywhere, whether it's Kafka, or Kinesis, or S3, or ADLS, or MQDT, or old mainframe systems, whatever it is, Spark can read it in.

Kind of typically, that's your kind of first hop into the data lake journey, is get it into one of these ingestion systems. It can be very simple. It can just be dumping JSON files into S3 or it can be running your own Kafka cluster. Kind of whatever makes sense for your particular use case. Really, here, I think you're thinking about how often is data arriving. What are my latency requirements? Do I need batching and things like that? You kind of decide there.

Then usually what I see now is people start by just dumping this data into Delta, and one of the kind of nice patterns that I've seen arise as people stop thinking about these kind of systems problems that I talked about before and they start focusing on their real problem, which is getting value out of their data, is they start developing a vocabulary for talking about quality. One vocabulary I've seen taken off quite a bit is this idea of bronze, silver and gold. So you have different tables with different quality classes. This isn't a specific feature of Delta. It's really just something that I've seen emerge after you kind of stop thinking about the other problems.

Usually, the kind of first hop in the data lake journey is this bronze table, and that's just a raw table where you read from wherever the data is coming from and you just dump it into this table with very little processing, if any. A lot of people say, "Well, wait a second. Why am I creating this copy of all of this data? How is that useful? Why shouldn't I be doing ETL?" There're actually a couple of reasons for this. I think the most interesting one is there're no bugs in a parser that you don't write.

If you don't do any ETL processing and you keep the raw data, you can always go back to that and re-compute. If you ever kind of find a type of analysis the you are thinking of, the raw data is there. There was a mistake in your next job. You can always go back and reprocess it. Then the other one is this thing that I was talking about before. It's very expensive in terms of human time to write these jobs and do data transformations. Bronze is a chance to just collect everything without thinking about the value.

After you've got all of the data there in your bronze table, usually start by trying to clean it up. Data is dirty. We know that's a fact, and so you need to kind of start filtering it, doing aggregations, augmenting it by joining with other datasets, normalizing things that you kind of have one consistent representation. This is where people start labeling these tables as silver tables. They are cleaned up. They've done all these kind of data munging, but they're maybe not quite ready for consumption. They don't answer a question that's important to your business.

Again, people are, "Wait a second. Why am I not just going to the report that my CEO wants?" Well, the silver tables are a kind of this nice opportunity. One, again, it's about kind of debugging. If you have this intermediate step materialized as an actual table and something looks wrong in your final analysis, you can actually go and use the full power of SQL to ask questions about this intermediate state. You can say things like, "Wait a second. How many distinct values are in this column and how many cases is this column null?" That really helps you understand where things are going wrong in your analysis.

The other reason why I think it's actually pretty valuable to materialize these intermediate states is that cleanup that you did is probably not only valuable to you. There's likely many other people in your organization who could use that as a starting point for that analysis. By creating that table, you kind of let them skip a lot of that initial work.

[00:23:54] JM: Just on the note of the bronze and silver tables. You're creating these intermediary tables on the way to cleaning up the dataset. Are you writing these intermediate tables into your data lake?

[00:24:09] MA: Oh, yeah. Yeah. All of these are present inside of the data lake, and really I kind of view the data lake as this continual process of taking raw data and turning it into something that is valuable.

[00:24:19] JM: Okay. Got it. I can see the value of sharing these bronze and silver tables with other people who might like these intermediary cleaned up datasets.

[00:24:29] MA: Yeah. Then after you've got those, then you kind of move on to your gold tables. This is the point where you actually start having kind of high-level business aggregates that mean something to somebody important. They actually are going to kind of drive the decision processes within your business.

Here, you can leave these in the data lake or people start pushing them to other sources. You might push them into your favorite dashboarding software, do a Tableau extract or something like that with it. You might push it into a more traditional data warehouse, or it's becoming increasingly common for this to be the final hop. This is actually where the most valuable data lives and this is kind of the single source of truth where people query it.

[00:25:05] JM: Okay. Now this is a workflow that I think is a little bit different than how people might think about data warehousing where oftentimes if you're thinking about a data warehousing workflow, you might have each of these intermediate data tables just be intermediate transformations or post- transformation states of the data in a data warehouse and you may just get rid of these from the data warehouse instantiation. How does this workflow that you've just described, this bronze, silver, gold table representation of moving data along towards a usable ideal? How does that compare to how data warehousing operations proceed?

[00:25:55] MA: Here, it's kind of interesting to ask the question why are these things different. My thought is it's really about scale and cost. I think people would like to do this same pattern inside of a traditional data warehouse. They could just never afford to pay that much money to Oracle or Teradata or whoever. I think that this is the magic of the data lake. It makes storing large amounts of data so cheap that you're allowed to materialize and persist these intermediate steps.

[00:26:24] JM: If we're talking about data warehousing, we should just talk a little bit about how Spark can replace functionality of a data warehouse for some users. We'll kind of gradually get to the idea of combining the data lake and the data warehousing functionality or how Databricks looks at that. But can you just talk about how Spark usage compares more generally to usage of a data warehouse?

[00:26:51] MA: Yeah. This is actually something that surprised me. When we looked at a bunch of customers of Databricks, we actually have found a lot of people who are using it as a data warehouse today, or at least they're doing the same kinds of workloads you would traditionally see on a data warehouse; reporting, BI, that kind of stuff, and that's something that has been steadily increasing kind of throughout my tenure at the company.

I think when I first started, Spark was really only an ETL tool, but with the advent of things like SparkSQL and the JDBC server, a lot of the performance improvements that went into Spark 2.0, it started to become a much more competitive kind of data warehouse solution. Now, I think there's a bunch of things that had only arrived relatively recently. I think ACID transactions are incredibly important. I think the ability to do DML, like update, delete and merge, that's table stakes for a data warehousing workload. Again, that's something that kind of Delta can bring to the table.

I think moving forward, at least the way we're thinking about this in Databricks, is there's a lot of value to having one single source of truth. But in order for that to really be true, we kind of have to close that gap between traditional data warehouses in Spark. There's a bunch of things that that means. I think it means that Spark needs to interoperate really closely with all these BI tools. We're working closely with Tableau and others to kind of make sure that that just works beautifully out of the box, that Tableau speaks the right dialect of SQL to get the best performance and all those kinds of things.

Working a lot on the kind of core performance of Spark, rewriting parts of it to be vectorized, take care of all of the tricks that we've learned in the last 10 years of database research. Then also worrying about things like concurrency, making sure that you can have lots of users querying the same table without having to spend a bunch of time partitioning them across clusters and things like that.

Again this is a place where I actually think Spark has a fundamentally better architecture than the systems that it's replacing. Decoupled compute and storage means that when at 9 AM when hundred business analysts show up at your company and want to start querying the table, you can actually just bring up three or four clusters and split them amongst that. Then at night when they all go home, you can stop paying for all of those clusters. Really, I think Spark came about at just the right time for us to see the cloud coming and to really kind of architect the system to work really well with the nice benefits of the cloud, in particular elasticity.

[00:29:15] JM: If we take an example of a popular modern data warehouse like Snowflake, for example. My interaction with Snowflake is going to be just like I am doing SQL queries over large datasets. But if I understand correctly, it kind of hides the implementation of a data lake plus a data warehouse under the covers. How does the experience of a user working with Snowflake compare to somebody working with Spark on top of Delta Lake?

[00:29:51] MA: Yeah. I mean, that's a good question. I think a lot of people don't even realize this, but I think with Spark SQL, that experience of kind of hiding a lot of the details about what's happening I think can be very similar. To me, the fundamental difference is whether or not you own your data. I think the biggest difference here is when you're working with something like Snowflake, you have to ETL that data into Snowflake. Snowflake owns it. Snowflake charges you for the storage of it, and you cannot query that data with anything other than Snowflake. If you want to query it with something else, you have to export first, and that's kind of an extra step to be done.

The biggest difference with Delta and Spark is you still own the data, and I think that's actually pretty important when you're going to collect very, very large amounts of it. We've been working with a bunch of other engines, because I'm obviously supervised. I think Spark is the best tool to query this, but for some use cases, people want to use Presto. They have legacy Hive jobs. They want to work with Apache NiFi to ingest data.

We're actually working with all of these communities to make it possible to have one single data lake built on Delta that all of these tools can queries. You really can choose the best tool for the job and you're not locked into any one particular solution.

[00:31:05] JM: Yeah, that seems practical and more economical in many cases, because the way that these data platforms have evolved, a lot of people have tons of data in HDFS already and maybe they want to – They could migrate all that data to S3 and make some of it available in Databricks Delta. But in any case, they may not want to copy all of their data into Snowflake, because if you copied all your data into Snowflake, it would be less economical I believe. Can you tell me more about the economics of legacy data lake infrastructure in contrast with one of these data warehousing systems like Snowflake or BigQuery?

[00:31:50] MA: Oh man! Yeah. I have to be honest. I don't know if I'm an expert on the particular pricing, but let me say what I think I at least understand. With the traditional data warehouse, you have coupled compute and storage. You're paying a bunch of money for compute even when you're not using it. You have to provision your entire data warehouse to operate at the peak of what you're expecting, which is pretty expensive if you have low utilization the rest of the time.

I think Snowflake, you pay for storage, and I think BigQuery pay for data that is scanned. That's at least a step better. If you have kind of big peaks of utilization that go down at certain times, I think that ends up being much better. To me, still, I think the best thing is get the lowest possible cost for storage, which is I think what cloud providers charge, and then only pay for the compute that you're actually using.

[00:32:41] JM: We should take a step back. I kind of launched us into a discussion of Databricks Delta, and data warehousing, and data lakes without talking about what you're actually working on. You lead the Databricks Delta team. Delta is a storage layer built on top of Spark. It uses parquet files in cloud bucket storage. What problems are you trying to solve with Delta?

[00:33:05] MA: Yeah. The one key problem is how do you build a scalable ACID transaction system on top of these things, and that is the core thing that's like Delta's magic trick. Then once you have ACID transactions, many things become easy. We have kind of idempotent streaming writes into Delta tables. Delta can also be a source for streaming efficiently from a table. Update, delete and merge these DML commands that modify the data kind of using

standard SQL semantics. I think all of these things become possible once you have that transaction log.

Basically, our first use case was exactly this. It was streaming into a giant table and just do that correctly with exactly one semantics and make it scalable, but once we have built that core, we then spent the next two years just adding all of these kinds of other features to really start making Spark look more like your traditional data warehouse.

[00:33:55] JM: I thought that data lakes were not typically used for transactions. I thought that every time you wanted to throw new data into the data lake, you just add a new file. You keep yesterday's files. You keep the files from 10 years ago. Why do you want transactional support in a data lake?

[00:34:15] MA: Yeah. I think you're kind of right there, except what I would basically argue is what everybody was doing with their data lakes before things like Delta existed, was they were just building poor man's transactions. A pretty common pattern here is if you're only appending, you're right. That works just fine. You just keep adding files and it's great. But if you have a failure while you're appending, how do you know if it actually happened or not? How do you know what made it out there? Then it turns out most data isn't static. Something like GDPR happens and you need to go in and remove somebody from your data lake. You want to do change data capture where you're getting kind of a feed of updates and you want to have your data lake reflect this operational system. Those things become very, very difficult.

As soon as you start modifying the data, you start doing kind of these like – I call them poor man's transactions. One trick is you use partitioning. Rather than have one giant table, you break your table up into a bunch of partitions by year, month, day, whatever kind of makes sense for your granularity of data. Then when you're doing modifications, you do them at the partition level. If you need to do an update to a partition, delete the whole thing, rewrite the whole thing. That works, because at least now you can reason about the correctness. You're no longer thinking about these partial failures. We have these nice boundaries for kind of containment of failures, But it's something that every single user of that table has to worry about.

I've seen so much Spark code that is deeply tied to the partitioning of a table and it's just really kind of violating one of the fundamental tenants that the databases came up with a long time ago, which is data independence. You want your data transformation logic to be independent from the actual physical representation of the table, and that just wasn't possible with these tricks that people are using. Anytime you want to change something, let's say all of a sudden you start getting a lot more data than you are getting before. Your company get becomes wildly successful and you need to change from month partitioning to day partitioning. Well, now you have to rewrite all of your applications, and that's incredibly expensive.

ACID transactions are just as really simple building block. They give you data independence, and now you can just kind of think about what's actually valuable, those data transformations and the domain that you're doing them in.

[00:36:29] JM: Make that explanation a little bit more concrete. Can you give an example application that explains why ACID transactionality would be useful for a data lake?

[00:36:41] MA: Yeah. Let me give one super concrete example. I mentioned this a little bit, but let me talk about it some more. GDPR, you have a massive collection of data and some user gives you a DSR, a data subject request where they say you need to remove all of your data that's relevant to me from your data lake and you have 30 days to do it or you are fined. I forget the exact numbers, but some crazy amount of money that every company wants to avoid.

Before ACID transactions, before Delta, I actually saw customers. The way they would do this is every 30 days they would copy their entire data lake, filtering out the users that had given them DSR's in the past. The reason for that is how else do you do this without possibly corrupting the table? If you just have files and you're modifying a file and you crash in the middle, you kind of don't know whether it happened or not. So it just becomes the bookkeeping of making sure that things worked even in the presence of partial failures in a distributed system becomes incredibly expensive.

Once you have Delta, once you have ACID transactions, that operation now becomes as easy as saying delete from where username equals Michael. You run that command. It deletes all the data and you're good to go. The magic trick here is what Delta is actually doing is it's

implementing this really well-known database trick called multi-version concurrency control. We never actually modify data in Delta. We only create a new copy of it. Anytime you're changing a single row in a parquet file, we just create a copy of that parquet file with that row modified. What the transaction log has is it just pointers to which parquet files are valid at any given time.

You can actually – This gives you a bunch of things. First of all, it makes all of those problems really easy, because it's that creation of the transaction log entry that atomically modifies a whole bunch of files in a single operation. It either happens all at once or not at all, and it also enables other cool use cases like time travel. You can actually now go back in time and look at old data. I really think it just fundamentally simplifies anything where you need to modify the data in place.

[SPONSOR MESSAGE]

[00:38:55] JM: Today's show is sponsored by StrongDM. StrongDM is a system for managing and monitoring access to servers, databases and Kubernetes clusters. You already treat infrastructure as code. StrongDM lets you do the same with access. With StrongDM, easily extend your identity provider to manage infrastructure access.

It's one click to onboard and one click to terminate. Instantly pull people in and out of roles. Admins get full auditability into anything that anyone does. When they connect? What queries they run? What commands are typed? It's full visibility into everything. For SSH, RDP and Kubernetes, that means video replays. For databases, it's a single unified query log across all database management systems. StrongDM is used by admins at Greenhouse, Hurst, Peloton, Betterment and SoFi Control Access.

Start your free 14-day trial of StrongDM by going to softwareengineeringdaily.com/strongdm. That's softwareengineeringdaily.com/strongdm.

Thank you to StrongDM for sponsoring Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:40:18] JM: You gave the examples of the bronze, and silver, and gold tables, for example. I might want to be recreating each of these tables once a day. I may want to be going through this process of creating a bronze table, a silver table and a gold table every day. If I'm using Delta Lake as the place to put each of these intermediate tables, am I overwriting the bronze table from yesterday when I create my table today?

[00:40:48] MA: Yeah, that's a really good question, and I would say you can. Some people do it that way. Delta would fully support that. You can do kind of nice atomic overwrites of tables or partitions if you want, but usually that's too expensive, and this is where I think the unification of streaming and batch becomes super important, because often what you want to do is it's actually not everything changes every day. It's just data is arriving incrementally, and Delta fully integrates with all of the APIs of Spark including the streaming APIs.

A much more common pattern here is you'll have your bronze table. You'll have a stream continually writing new data to that bronze table. You'll have another stream that's reading from bronze and writing into silver doing that extra cleanup and whatnot, and then finally you have another stream that's maybe doing aggregation and other things, a stateful stream that's reading in the silver table and writing out the gold table. There's a bunch of nice properties here, but I think the biggest one is streaming dramatically simplifies this work.

If you think about what kind of most ETL pipelines are dealing with, they're thinking about this problem of what data is new since I process last. How do I take that data, process it, push it downstream, transactionally guaranteeing exactly one semantics. How do I checkpoint where I'm at in case there's a failure so I can pick up from where I left off? All of these things are happening automatically with streaming.

What I really think, I kind of try to correct this misconception pretty often. People hear streaming and they think, "This is only for hyper low-latency use cases where I am willing to pay a whole bunch in terms of money and maybe also complexity in order to have low-latency," and I think that's wrong. I think streaming is for any query where your data is changing and you want to compute the new answer to that query incrementally and efficiently.

[00:42:35] JM: Very interesting. Now, my intention there was to better understand how the ACID semantics would be useful for this bronze, silver, gold example. You gave a very different answer, which is pretty interesting, but is the ACID semantics relevant here if we are talking about this example?

[00:42:56] MA: Yeah. Maybe not to harp too much on this, but the only reason streaming works is because of ACID semantics. The way that you get exactly once in Delta is by leveraging ACID. We kind of play this cute trick where when we're writing into a table, we both write the data into the table and we atomically record in the metadata of the table which batch is being added. What collection of data is there? That allows us to make this write idempotent.

When you combine that idempotent with determinism in the input, you get exactly one semantics, but we only get that because we have ACID transactions. I guess my answer is exactly what I said, it is still ACID transactions that were under the covers.

[00:43:42] JM: I see. If we think about this naïvely, like if I just use like a Python script, some naïve Python script to pull data out of S3 and then write somewhere else in S3, why is that not ACID compliant?

[00:44:00] MA: Well, in S3, it's extra complicated, because S3 is only eventually consistent. I think if you're just – Again, if you're just appending data, everything is much simpler, but that's just never how it works in the real world, because you write some data out, you crash in the middle and you don't know if it's there or not. Let's look at a slightly more complicated example where you're actually modifying something. You have a directory. You've written some data out. You want to change the data that's there. So you delete some files and you write the new copy there. The problem with S3 is for an arbitrary amount of time and actually up to hours, both sets of data are flickering in and out of existence. If you read that table, you might get the new answer, you might get the old answer, you might get some combination of the two of them. I think that just makes it very difficult to reason about correctness.

[00:44:50] JM: Tell me more about what was built in the data lake transactional system that alleviates that uncertainty between the pre-write and post-write values.

[00:45:03] MA: Yeah, that's a great question. There're a couple of things. One is we get around eventual consistency by never reusing a file name. Every file that's created in Delta has a unique GUID that has never been used before, and S3 does guarantee read after write consistency for the creation of new keys. In terms of this, S3 is kind of the weakest. HDFS, ADLS all give you these things just out of the box.

The other magic trick that Delta brings here is atomicity. When you are writing out – So let's say you're beginning a transaction. You're writing out to a table. You write out a bunch of files, but just because they are present on S3 doesn't mean anything. They're just kind of – You can think of them as like a staging. They're just waiting to actually be committed. What actually adds them to the table is we atomically create an entry in the transaction log that says, "Hey, these files are now valid."

What's really cool about that is you're adding a whole bunch of files at once. You actually kind of get this cross file atomicity, which makes everything easier to reason about, because now that operation, this like, "I want to do ETL. I want to take the last hour of data and add it to the table." That all happens or all doesn't happen, and after it's done you go and check and say, "Hey, did that happen or not?" If it didn't happen, you could do it again. If it did happen, you can say, "Oh, okay. I can skip this. It already worked."

[00:46:22] JM: Really, a lot of this is built around the transaction log that you're maintaining.

[00:46:27] MA: Yup. Really, if I had to describe Delta in one sentence, I would say it's a transaction log on top of Parquet files.

[00:46:33] JM: Right.

[00:46:34] MA: And all the cool things you can do once you have that.

[00:46:36] JM: If we contrast this with patterns around Kafka, your example of the bronze, silver, gold tables that people might be preparing, it sounds somewhat similar to how some people use Kafka in terms of data enrichment or data cleaning like where they'll write a bunch of raw data into one topic in Kafka. They'll clean it and write it to a second topic and then they'll

enrich it and write it to a third topic and they'll use that third topic for some application or they'll write the third topic to their data lake. There're all kinds of patterns that people use here. How do you see the usage of Delta Lake comparing to the usage of Kafka topics?

[00:47:28] MA: Yeah, that's a great questions. I think, first I'll say, I often see these things used together. A pretty common pattern is we use Kafka as a buffer before Delta. People write a bunch of data in there in kind of tiny little packets and then Spark takes reasonably-sized batches every couple of seconds and writes them into Delta. That's a great architecture that works pretty well.

To me, the really biggest difference is between these two systems are cost and elasticity. I think Kafka is great for acting as that buffer. It has a much lower latency for writing into it than something like Delta. Creating a Parquet file is pretty expensive. You're going to be paying hundreds of milliseconds to do that. I guess really the biggest difference is kind of throughput versus latency.

Delta is optimized for storing massive amounts of data, so years, and doing it with very high-throughput. But the latency of any given operation is in the hundreds of milliseconds are the minimum. Usually seconds to minutes.

[00:48:27] MA: Kafka in contrast is great for very low-latency processing. If you want to ingest data in milliseconds, I think it's very good at that, but you're typically not storing years of data in Kafka. You're storing hours, days, maybe weeks, but I think even then you're starting to push it. Again, Kafka is a kind of couple compute and storage cluster. If you're running Kafka in the cloud, you are maintaining those instances all the time. I think you're worrying about things like scaling up and scaling down. In contrast, your data lake is inherently elastic. You only bring up clusters when you're using them. When you're not, you shut them down and you let S3 worry about utilization and all those other problems.

[00:49:06] JM: Tell me if this is an inaccurate way of looking at it, but it seems like the reason that you have kind of a divergence in what's useful for Delta Lake versus what's useful for Kafka is Kafka I believe is in memory and on disk, or a lot of the data in Kafka at least is in-memory and then some of it might only be on disk. I'm not sure, but it's in memory and on disk.

Spark is only in memory. Delta Lake is only on disk. You just might have different patterns around how these different systems are used, because you have one that Spark entirely in memory. Delta Lake entirely on disk, and then Kafka which is both.

[00:49:48] MA: That's not incorrect. But to me really I think it's just about whether or not you are optimizing for throughput or latency. I just think you end up building very different systems depending on which one you care about.

[00:49:58] JM: Tell me more about the development of Delta Lake. What were some of the difficult engineering problems in building it?

[00:50:09] MA: I think the biggest engineering problem as soon as you start doing storage, it's just very important that you don't get it wrong, because if you ever mess up storage, there is some persistent state out there that reminds you of your mistake forever. I'll give you an example of one. We actually ended up catching pretty quickly before it affected too many people. We're able to fix it. But at one point there was a mistake in our statistics calculation.

If you had a string that was over 32 characters, we were incorrectly truncating it. I'd say, "Well, why does that matter?" It turns out we use statistics in a bunch of places for correctness, and so if the statistics are wrong, the answers to your queries can be wrong.

I think the biggest challenge is how do you do validation here for a system that's going to be storing petabytes of super valuable data? We have a pretty wide variety of techniques here. For that particular statistics bug, we added a whole suite of random query testing that just issues a whole bunch of different types of queries that write and read from Delta and then does the same thing against normal Spark. This was really cool, because we actually found a bunch of bugs in Spark and a bunch of bugs in Delta, but anytime they disagree, something is definitely wrong.

Another class of problem is just how do you make this thing robust so that it processes all the time without failing. Here, there's a bunch of kind of weird tricks. An example here is this is a weird quirk of S3, but if you ask S3, "Does this file exist?" before you create that file. It doesn't, and they return no. They actually that cache negative answer.

Even if you create the file for an arbitrary period of time after you create the file, they'll still tell you it doesn't exist, which causes Spark to crash the job and say, “File not found.” We just spent a whole bunch of time going through just to make sure that Spark didn't have any cases where it ever asked for a file before it was created to make sure we never primed the negative cache, right? That’s silly, but it was super important to the robustness of Delta to keep it running. For that kind of stuff, my answer is eat your own dog food.

Of course, at Databricks, we’re a cloud company. We have a whole stream of metrics coming from all of our customers to understand how they're using Spark and how they're using the product. Where they're having problems? One of the things we did really early in the days of Delta was we just created a complete copy of our data teams pipeline that was running on Delta and streaming and using all of this stuff. I think that was just a great way to not only find these kinds of critical correctness bugs, but also find usability bugs in the system as well, like, “Oh! Hey, that was really confusing, or I don't understand why the performance of this is really bad.” When you deeply use your system for a problem you care about, I think you end up with a much better system as a result.

[00:52:57] JM: I'd love to know little more about your perspective on the competitive landscape. The most common data warehousing systems that I hear about are Snowflake, Red Shift and BigQuery, at least if we’re talking about very modern data warehouses. If somebody is deploying a new data warehouse system today, they’re probably deploying one of those three or Spark. Those other systems are closed source. What do we know about how these systems differ in their design and do you take any inspiration or deliberate differences in your own design of building what is essentially for many users going to be a data warehouse?

[00:53:43] MA: Yeah. I guess with those three, I would probably start by segmenting them into coupled compute and storage and decoupled compute and storage. Red Shift spectrum is kind of a middle case, but Red Shift, the normal Red Shift I think most people use is coupled compute and storage, which make some performance things easier but also means that you don't have elasticity. If your load scales up, it’s quite a bit different.

In contrast, I think things like BigQuery and Snowflake started off cloud native, or BigQuery I guess. I don't know if that was like pre-cloud, but Google's kind of always been a cloud internally. They started off with this idea of the storage is over there, the compute is over here. I think at least architecturally they end up looking a lot more like Spark and Delta. But then again I think the real question is do you own that storage and can you query it with other tools or not? There, I think you kind of correctly identified that. I think that's the biggest differentiator with Spark and Delta, is that even if you decide you don't want you Spark anymore, you don't have to export that data or do any extra ETL. You could just use other engines to work with it.

[00:54:50] JM: All right. As we begin to wrap up, we had a show recently about Ray and Anyscale, which was cofounded by Ion Stoica, who also worked on Databricks in the early days. Do you have a perspective on where Spark applications and Ray applications might differ?

[00:55:07] MA: Oh man! I might have to be honest. I know almost nothing about Ray.

[00:55:11] JM: Okay.

[00:55:13] MA: I hope Ion isn't listening to this.

[00:55:16] JM: That's totally fair. Okay. Well, different kind of question. When you kind of to take the perspective of Spark as being this thing that's centered more of a data platform, this decoupled data platform. Do you think there's a certain type of business or a certain type of customer that might feel more residence if you just described Databricks as a data warehousing company?

[00:55:44] MA: Oh man! You're like listening in on some of our meetings.

[00:55:48] JM: The marketing meetings? I pass these billboards. I mean, I'm sure you pass these billboards all the time. It's like the Snowflake billboard and then the Databricks billboards. Snowflake loves to use the word data warehouse.

[00:55:59] MA: Yeah, that's a great question. To me the bigger question is what are the things that are missing to make us look more like a data warehouse? I think it's actually something that

we're actively working on. I think kind of areas of focus for us need to be how do you make sure that the SQL experiences is just absolutely flawless, first-class, everything works really easily? How do you make sure the optimizer is up to snuff so that SQL queries run very quickly? How to make sure all of the other kind of things that exist around the data warehouse are there? Things like ACLs and governance, right? Those aren't like directly Spark things, but they're very important as soon as you start storing important data in it and get something that Snowflake does pretty well.

My take has always been like, "Hey, we're actually cooler than the data warehouse, so we shouldn't use that term." I think it's absolutely fair to be comparing that, and I think focusing on what it takes to make us look more like a data warehouse is definitely something which we're spending a lot of time on.

[00:56:52] JM: All right. Michael Armbrust, thanks for coming on the show.

[00:56:54] MA: Yeah, thank you so much for having me.

[END OF INTERVIEW]

[00:57:05] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product

development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[END]