EPISODE 1108

[INTRODUCTION]

[00:00:00] JM: Pandas is a Python data analysis library and an essential tool in data science. Pandas allows users to load large quantities of data into a data structure called a data frame over which the user can call mathematical operations. When the data fits entirely into memory, this works well, but sometimes there is too much data for a single box.

The Modin Project scales Pandas workflows to multiple machines by utilizing Dask or Ray, which are distributed computing primitives for Python programs. Modin builds an execution plan for large data frames to be operated on against each other, which makes data science considerably easier for these large datasets.

Devin Petersohn started the Modin Project, and he joins the show to talk about data science with Python and his work in the Berkeley Rise Lab.

If you are interested in sponsoring Software Engineering Daily, send me an email, jeff@softwareengineeringdaily.com. The show reaches 30,000 engineers each day. And if you are interested in reaching those engineers as well, we'd love to hear from you. You can also become a paid subscriber to the show by going to softwaredaily.com and clicking on subscribe. That would help us support the show and it would mean a lot to us. On softwaredaily.com, you can also find all of our old episodes as well as episodes that are related to topics that you're interested in, and you can download the Software Daily apps for iOS or Android, which also contain all of our episodes.

[SPONSOR MESSAGE]

[00:01:38] JM: SAP is a company that touches $23 trillion of consumer purchases around the world. You've probably heard of SAP, but you may not know about SAP's open source investments, such as SAP Data Intelligence. SAP Data Intelligence connects and transforms data to extract value from the distributed data landscape and embraces the best of open source technology.

SAP Data Intelligence brings together data orchestration, metadata management and powerful data pipelines with advanced machine learning enabling close collaboration between data scientists and the rest of your infrastructure teams. SAP Data Intelligence runs on Kubernetes. SAPs contribution to Kubernetes includes the Gardener Project that helps to operate, monitor, manage and keep Kubernetes clusters alive and up-to-date. To learn more about SAP Data Intelligence, you can visit sap.com/sedaily. That's sap.com/sedaily.

[INTERVIEW]

[00:02:48] JM: Devin, welcome to the show.

[00:02:50] DP: Thanks for having me.

[00:02:52] JM: We are sitting, what is it? Day 90 or something of quarantine, and talking to you over Zoom. You have a virtual background that is your face. It's very endearing. And we'll talk about Modin and data frames today. Your recent research has focused on data frames, Pandas especially. Pandas is a famous data analysis library in Python. It's seen as an essential tool in data science. Describe the data science landscape in terms of the available libraries and frameworks today.

[00:03:29] DP: Yeah. We live in an interesting time when it comes to the data science landscape, I think. A lot of the tools that are really common that are taught in schools, college, things like that, those are the tools that are for smaller sale data, and a lot of the time it's because these courses are not designed around teaching students to use big data, of course.

There are also tools on the large scale, and there's a big rift between these two sections. You have tools that are good, really good for smaller scale data and they're really intuitive. They give you a real feel for having this kind of grasp of your data. At the large scale, you have these really big batch processing engines that kind of put all of these extra restrictions on what you can do, because there are a lot of things that are kind of challenging to scale that these smaller scale tools do.

So we get this kind of rift in data science, where a lot of companies have these pipelines where they start in Pandas, or they might start in Spark and run a batch job to kind of filter down their data so that they can use it in Pandas. And then they use Pandas to kind of do some preliminary analysis and understand the data. And then they hand it off to somebody else to rewrite that workflow into Spark. And this rift here is happening because there're different requirements for different scales of data. I think data science, it's in kind of a – I don't know if weird is the right term for it, but it's in an interesting place where we don't necessarily have the right abstractions for performing data science at all scales using the same kind of toolset.

[00:05:15] JM: That workflow sounds pretty painful. The workflow you're describing where I've got a huge dataset and I'm going to use Spark to whittle it down to a manageable dataset so that I can work with it more easily on perhaps my laptop. And I would like to know more about why is that kind of workflow necessary. Why couldn't I just do exploratory data analysis in Spark or – Yeah, I guess Spark would be the best example that I can think of.

[00:05:53] DP: Yeah, definitely. So you could do this, but the problem with tools like Spark at that scale is that they require the user to be very intimate with the running environment that they're operating in, and also with a data layout. Spark put some restrictions on requiring to understand kind of partitioning and trying to understand when do you want to trigger computation. The kind of lazy, the lazy evaluation paradigm.

I mean, Spark is a great tool, and I do not have anything ill against Spark. But what we're seeing in the real-world and a lot of business applications is actually that the typical Pandas user is not an expert in distributed computing. And Spark is a tool that kind of requires you to have this distributed computing understanding to not be penalized for your performance. If your partitioning is bad, for example, a lot of times you'll need to throw in some repartition calls in the middle or if you have like a really hot key, for example, and you do a group by on that key, you end up with very extreme data skew.

This partitioning is so important in performance that if you don't get it right, you end up paying very, very heavily. And I think that there's also a big discrepancy between the APIs as well and the capabilities. Spark can't really do everything that Pandas can do. When you use Pandas, you get this feel that you're controlling your data. The system is not controlling you. You are controlling the system. You are controlling the data. You can wrangle it. You can do almost anything with the data in Pandas.

Spark, because they're working on this kind of more relational model, more of a database style model, they put some restrictions on kind of what you can do, because those are the things that are really easy to scale in batch processing systems. I think there's a big of a mismatch between Pandas and these batch processing frameworks where batch processing [inaudible 00:08:03] synchronous processing, it's not necessarily the right paradigm for all of your data frame operations.

[00:08:12] JM: Let's dive deeper into that term, data frame. What formally is a data frame?

[00:08:20] DP: Yeah. This is part of a lot of the work that I've been doing lately in the RISELab actually. So the data frame before we started working on it hadn't really been formalized, and there's a lot of kind of competing implementations, you can say. Things that consider themselves data frames. From our perspectives, data frames are very specific and distinct structures specifically distinct from relational tables, or databases, and matrices.

So data frames kind of have four main components. They have an array of data, or you can consider it as a kind of heterogeneous matrix. So you can have columns that have different types of data. They have row labels and they have column labels and they also have column domains or column types. The column domains are interesting here, because the real distinction with relational systems is that in a database, every operator that you do needs to have a known output data type for any given input data.

So if I give you a table of floating point number, then I know that I can determine in a relational system what the output data type will be for any given operation that I do. In data frames, this restriction is not there. We don't have this restriction. So that allows us extreme flexibility and being able to kind of define operators that don't have necessarily a known output data type. And the real power here comes from being able to kind of have this f a Two. Kind of have this flexibility, this control over your data that I was mentioning earlier. The data frame is ultimately extremely useful for data that you don't necessarily know what your input data types are. You don't necessarily know where are your bad values. Where is that rogue string in a column of integers? We need some kind of tabular structure that lets us handle these without completely blowing up. And databases are not well-suited for this kind of schemaless, semi-structured, kind of unstructured data.

So a lot of my work in my PhD studies has been around this idea of let's define this data frame and actually use it for what databases can't really do and make sure that we have this well-formalized basis for any kind of implementation that we end up doing.

[00:10:59] JM: So the data frame advantage has to do with the looseness of the output typing?

[00:11:10] DP: Not exactly. It's more that I don't need to know the data types of any particular column. So let's take a concrete example. Let's take transpose, which is something that we have focused and studied a lot. Transpose is relatively commonly used, and it kind of depends on your interpretation of transpose, because there is such a thing as a relational transpose. But our interpretation of the data frame version of transpose is this idea that you can basically just convert the columns into rows and rows into columns and interpret them that way.

If you look at data in the wild, data in the wild comes in all kinds of formats. And if you're getting data from a data source that places these data in a row format when you really need them to be in a column format, in those cases, transpose is vital to the entire usage. So there might be, for example, a case where you have a table that is laid out in a certain way for human consumption. It's much easier for people to compare different items if they’re in columns.

So let's take an iPhone example. So if I have an iPhone, a set of iPhones, and I have a set of qualities, it’s much easier for me to compare the different properties of the iPhones if the iPhone, if the individual iPhones are in columns. You'll see this on Amazon webpages all the time. Comparing these different properties is much easier for human consumption when I’m looking at a webpage. But what I'm doing an analysis, it doesn't make sense for the data to be laid out that way. I want my properties to be the columns, because those are the common attributes for all of my iPhones.

So transpose is an important data wrangling tool. And not only that, but it's also important in a lot of operators that people use in some of the work we've done in kind of defining this algebra.

So the output schema for a transpose, getting back to your question about schema, the output schema of a transpose cannot necessarily be known given an input schema. And that I think is the real power of being able to kind of define a schema that the system can basically induce after an operation has completed. Instead of having to know that my output column is going to be all integers before I even do any data processing. So that's a very concrete and big distinction between data frames and databases.

[00:13:49] JM: Can you go a little bit deeper into that? A data frame versus a relational table, like why can I not think of a data frame as simply a big relational table, a materialized view?

[00:14:04] DP: Data frames – Yeah. I mean, this is a great question. And I think that a lot of – If we can kind of backup, a lot of people do choose to interpret the data frame as a relational system, and they've implemented systems that kind of pick and choose some of the properties of the data frame so that they can scale them. But looking at the differences, the main differences between relational systems and data frames, I mean, one of the key differences is order. Data frames have an implicit order, and relational tables or databases do not.

So if I choose to reorder rows, that produces a new data frame. But in a relational table, there is no concept of order. So reordering rows can be done by the system at will if it wants to do some optimization underneath of the hood or something like that.

Another important thing here is the labels. When you're working with this data, if you really want to have a good grasp on your data labeling your rows, labeling your columns, it's very important. Now, databases do have column labels, but they don't have this concept of a real label. And the row labels are very commonly used in Pandas and it is one of the most kind of keystone features that makes Pandas so good at wrangling this data.

The lazily induced schema, we kind of already touched on. There's a lot there. And I think kind of the last major difference is the fact that data frame support linear algebra operations as well. So it's much easier for me to take a Pandas data frame and pass it to scikit-learn for example, and scikit-learn can then interpret it and kind of treat it as if it were a matrix without me having to kind of convert it to some other system. And there are, of course, machine learning libraries that work on relational tables. But the idea that you can use one system to kind of clean your data and then pass it off directly to a machine learning library, that I think is the kind of one of the biggest pluses of a system like Pandas.

[SPONSOR MESSAGE]

[00:16:26] JM: Scaling a SQL cluster has historically been a difficult task. CockroachDB makes scaling your relational database much easier. CockroachDB is a distributed SQL database that makes it simple to build resilient, scalable applications quickly. CockroachDB is Postgres compatible, giving the same familiar SQL interface that database developers have used for years.

But unlike older databases, scaling with CockroachDB is handled within the database itself so you don't need to manage shards from your client application. Because the data is distributed, you won't lose data if a machine or data center goes down. CockroachDB is resilient and adaptable to any environment. You can host it on-prem, you can run it in a hybrid cloud and you can even deploy it across multiple clouds.

Some of the world's largest banks and massive online retailers and popular gaming platforms and developers from companies of all sizes trust CockroachDB with their most critical data. Sign up for a free 30-day trial and get a free T-shirt at cockroachlabs.com/sedaily.

Thanks to Cockroach Labs for being a sponsor, and nice work with CockroachDB.

[INTERVIEW CONTINUED]

[00:17:48] JM: So, I've got this big dataset that I instantiate in a data frame and I want to do some them operations over that data frame. That works fine if it's a smaller dataset, but there are some limitations when it comes to scale. Can you explain the limitations around Pandas data frames?

[00:18:18] DP: When I say these things, of course, I don't mean anything ill towards Pandas. Pandas is the most used data frame by at least an order of magnitude. The implementation limitations I think are well-known by the developers. One of the biggest ones is the API, and it's very, very hard to kind of learn from the outset.

If I'm in a new data scientist who is coming from some scientific field, not necessarily computer science, this API that the Pandas has is over 240 operators per data frame alone. If you add in all of the other kind of utilities and series, which is the one-dimensional kind of object that Pandas has, all of these combined is over 600. And 600 operators, to learn for somebody who's just kind of stepping in and getting their feet wet with data science, it's daunting. It really is.

Here at Berkeley we have a data science team, or a data science teaching team and data science program, and one of the things that that they've done is kind of try to drill down pandas to get some –For teaching, to get some subset that is teachable to get. Basically it's a library that sits on top of Pandas that just kind of only exposes a subset. And this is really nice for people who were kind of coming in off of their scientific backgrounds not really having done a lot of programming in Python or any other language for that matter.

Another really key, I guess, limitation with Pandas is it's a single-threaded implementation. If I want more speed, no amount of hardware that I throw at Pandas will make any sort of difference. Effectively, I'm basically just limited by its implementation. I think these are the two kind of largest downsides with the Pandas implementation, and there are things that I've tried to tackle in my PhD studies as well.

[00:20:18] JM: Why is there this limitation on the data size of Pandas? It’s often said that memory should be – The memory in your machine should be 8 or 10 times the size of your data frames. If you've got a 1 gigabyte data frame, you should have 8 gigabytes of RAM. Why is that? Why do you need so much memory to deal with these data frames?

[00:20:46] DP: A lot of this boils down to kind of how Pandas was implemented and the internals of Pandas. Pandas was created to give you this this deep control over your data. Basically, if you want to update a single point in your data to kind of claim it. For example, changing a single rogue string to an integer or something like that, Pandas gives you this power.

The downside with this really tight integration of the API layer with the physical implementation is that you do have a lot of copying. So if you do some small operation, like, for example, you run a fillna, which fills all non-values with whatever chosen value. That will create a full copy even if there's only one null goal in the dataset. So a lot of this comes down to basically the really tight coupling between the API layer and the physical layer.

[00:21:48] JM: Can you tell me about these hardware limitations? So the integration between the software and the hardware layer. like I guess this gets into more around general limitations of computing, of at least single node computing. Tell me more about the limitations in terms of memory size when it comes to Pandas.

[00:22:12] DP: Yeah. A lot of the limitations around Pandas in terms of memory is coming from the amount of copying that they’re doing. And a lot of the Pandas internals are actually using NumPy. NumPy is a very, very commonly used numeric computing library and array computing for Python. And so what the developers of Pandas have done is kind of add some extra features and add some really nice abstractions on top of this kind of underlying array structure to make it effectively support the heterogeneous types.

The challenge a lot of times with the memory is that datasets are constantly growing. And we see this more and more as more companies try to process more data. And Pandas, unfortunately, because of this tight coupling between kind of the implementation of the API and the actual implementation of the physical data manipulation, this tight integration has kind of prevented them from being able to scale Pandas with the rest of the world, with the data that is being scaled out in the real world. This really tough copying problem in Pandas is something that they've been working on for many years, and it's really kind of a hard problem due to kind of the tight coupling that they present to users.

[00:23:51] JM: Why do Pandas data frames need to be copied so rigorously, so frequently?

[00:23:56] DP: They don't. It's not necessarily that this is a kind of requirement of the API itself. It's more of kind of an implementation where, effectively, one of the main challenges with Pandas is that the underlying arrays that back the data frame can be physically manipulated by the user if they access it via these kind of protected, semi-protected because it's Python, right? But these protected fields of the data frame itself.

So they can actually manually access. You can conceptualize this as they can manually access memory underneath of the data frame and modify that memory underneath the data frame to modify the data frame itself. This tight coupling is problematic for a number of reasons, but that is kind of the biggest problem with copying, is that if you share some memory address space for things that aren't modified but somebody goes in and edits manually that address space. Then all of a sudden, the changes are propagated across all data frames that share that address space, but the user wasn't expecting that, right? Because the user is expecting basically that to only affect the data frame that they've been touching. And so this is why the tight coupling between the API layer and the physical implementation is something that's so kind of challenging to fix in Pandas itself.

[00:25:41] JM: Yeah. I think it's worth talking now about how Pandas queries are executed, because if on every query or every Pandas manipulation I am copying my data frame, that's going to be expensive. Probably there are some alternatives here. Let's just explore how Pandas queries get executed.

[00:26:07] DP: Pandas along with most of the other py data stack generally fall under what we've been calling eager execution, which is basically executing operations as they come along. The downside to this with Pandas specifically is that you can train a lot of operators together, and then each of these operators is executed in the order that they've been called. You may never need to see any of the intermediate objects that were created. But if Pandas is creating a copy at every single one of these stages worst-case, then what you end up with is a single line of code that blows up your memory when you may have been at 5% or 10% before everything started.

So this eager execution challenge is – Since you can't do query planning, you can't know what the user is going to ask for next. The problem is that you end up materializing everything. Everything must be materialized before the next step can continue. I mean, there are a lot of downsides to materializing every single intermediate object when it comes to memory. These queries, generally, they're not extremely simple queries. A lot of times there, there are some complex joins or some expensive group good bys. And if you're doing a filter after any of these expensive operations, effectively, you don't need to do the whole join or the whole group by.

What you need to do is just filter beforehand, because you're never going to actually use that intermediate cross-product or giant join. So the eager evaluation kind of reliance, it's a hard problem, and it’s a user interactivity problem too, because users are expecting things to be evaluated as they’re submitting them. And this creates a whole bunch of interesting research problems from like the kind of from my standpoint, from the kind of academic standpoint, there a lot of interesting research problems around optimizing for this user experience, this user activity, and trying to avoid materializing every single intermediate object while still giving users this feeling that they're both controlling their data in kind of an intimate way, and also that that the CPU is working for them effectively.

[00:28:37] JM: Are there opportunities for query optimization under the hood? Would that be – If we could optimize the queries against these data frames, would that improve the constraints on memory, this copying problem? Would that be amenable to more scalable data frame usage?

[00:29:05] DP: Absolutely. It would. Memory is really only one kind of piece of all of this. But feel like memory is kind of the biggest pain point. And I'm glad we’re hitting on this, because a lot of people, whenever they kind of evaluate pandas, they show plots of like CPU times, basically. The query planning involved in data frames so far has been around basically transitioning from an eager evaluation setting to a lazy evaluation setting. The challenge with this in an interactive setting is that when the user is sitting at their keyboard and thinking about things, the CPU is not working towards an answer. The CPU is idle waiting for the next input. And in these lazy evaluation systems, they basically have tried to kind of move back to the declarative nature of SQL.

So in SQL queries, what you do is you write out a whole query and then submit it all at once. And the systems gets to kind of look at the whole query. Understand this is my world until I'm finished executing this query. I can rewrite things. I can manipulate things as much as I want. Lazy systems have basically taken that kind of declarative approach where they're basically queuing up operations. And then when the user does something that triggers computation, manually, for example, then those systems will basically look at the queue that they have and be able to rewrite things and treat that as their world.

So lazy is effectively declarative, and that kind of puts some challenges on the user who is trying to interactively debug their queries. So I think the hardest part about interactive data science and exploratory data analysis is this idea that when we are interacting with the system, a lot of times we sacrifice a lot of user time in the interest of CPU time. If you consider this lazy evaluation approach, what I'm doing is effectively queuing up things and keeping the CPU idle all the while the user is thinking, right? The user might be thinking for a long time. There might be these kind of long pauses in between when they actually submit queries.

In the eager evaluation approach, the CPU will effectively be working toward a result during that think time and the user then must wait until the eager evaluation system has finished doing its previous –Computing its previous query to move on to the next step. With this interactive model, interactive user model, there are a lot of really, really interesting challenges around kind of optimizing for the user's time, and that's been my kind of focus during my PhD program and my focus with the data from system I've been building, Modin. The user's time should always be more important than the system's time.

Really, as computer scientists and as software engineers, we think about systems in terms of the system, right? When we benchmark things, we compare against systems and we compare runtimes and things like that. But when the user is using the system, there is a lot more that goes into the amount of time it takes a user to go from the beginning of their workflow to the end, especially when it comes to interactivity and exploratory data analysis.

I think this space is really kind of – It's very new, I think, and there is a lot of really interesting research to be done in this idea of optimizing for the interactive user. And does it really matter if our CPU did not get to the answer as optimally as perhaps a lazy system if we've saved the user time overall? I think no. I think the right way to think about this is to treat interactive users as kind of separate from this batch case, which is where lazy really shines.

[00:33:26] JM: Okay. So if we make a lazy evaluator for Pandas operations, how does that change the execution of these Pandas operations?

[00:33:45] DP: Great question. So we can take a kind of a real world example. Dask, for example. Dask is a system. Dask data frames is a system that kind of try to take a subset of the pandas API and scale it as much as possible. And it's basically a system that enables lazy evaluation and query planning. And the way that it works, if we wanted to make Pandas lazy, it would be something quite similar, where, effectively, every query does not return the new data frame. It returns kind of a lazy object. And then when we actually feel like we're ready to trigger computation, then we can just basically do something like a .compute. And that will basically tell the system, "Okay. I'm ready for things to be computed now."

There are a lot of challenges with exposing the lazy evaluation semantics to users who are maybe not super familiar with how systems work, how computers work even. The domain scientists who are the primary users of Pandas, who come from non-technical backgrounds and aren't primarily computer scientists. What I've seen in a lot of notebooks from these people is that they end up triggering computation every couple of steps, which really limits the amount of query planning you can do if you're triggering computation after every step or every other step. And so, really, the challenges here are both from the user side and also from system side.

[SPONSOR MESSAGE]

[00:35:31] JM: Today's show is sponsored by Datadog, a modern, full-stack monitoring platform for cloud infrastructure, applications, logs and metrics all in one place. From their recent report on serverless adaption and trends, Datadog found half of their customer base using EC2s have now adapted AWS Lambda. They've examined real-world serverless usage by thousands of companies running millions of distinct serverless functions and found half of Lambda invocations run for less than 800 ms. You can easily monitor all your serverless functions in one place and generate serverless metrics straight from Datadog. Check it out yourself by signing up for a free 14 day trial and get a free T-shirt at softwareengineeringdaily.com/datadogtshirt. That softwareengineeringdaily.com/datadogtshirt for your free T-shirt and 14-day trial from Datadog. Go to software engineeringdaily.com/datadogtshirt. Thank you, Datadog.

[INTERVIEW CONTINUED]

[00:36:42] JM: Let's get into Modin, which is what you are working on. Explain how Modin came to be. Why did you start working on this?

[00:36:52] DP: It started effectively as kind of a weekend project, if you will. I was approached by Ion Stoika, who is the director of the RISELab and who – He's one of the main PIs on the Ray project. I was approached by him and asked, "Can you see what you can do with the data frame?" Ray didn't have a data processing engine at the time. And so he approached me to kind of see what I could come up with effectively and kind of propose something.

So I kind of took a little bit of time. I hacked something together and then I released a blog post. And the blog post, it got pretty popular, I would say over the weekend. And then all of a sudden it became my primary project. So, it's interesting, because it's kind of accidental, right? I know the problems in the area, and I came from working on genomics and kind of working with data scientists who are researchers in these really large-scale DNA analysis pipelines. And that was my background. So I understood the kind of domain scientists viewpoint on systems and I basically just translated that to Modin, which effectively you know abstracts away all of the complexity from the user. Users are – If we treat users as the system we want to optimize for, users are effectively the most important part of the entire data science pipeline. And so if you optimize for the user and the user's time, then that starts to open up a whole bunch of really new and kind of untouched research problems that I've been exploring.

[00:38:53] JM: How does Modin improve the performance of data frames?

[00:39:00] DP: Modin has basically taken kind of good software engineering practices and some novel kind of under the hood optimizations to scale Pandas. So Modin is abstracted out into multiple layers just like you would want in any system. And the Pandas API basically sits at an API layer. And the nice thing about that is that we can start to explore other APIs that people are using. SQL, for example, and apply those to a data frame context, because data frames are these superset of SQL. So we can start to kind of have this unified execution backend that is highly optimized and be able to expose different interaction modalities to users to kind of interact with the data frame system and whatever API they're comfortable with.

So, underneath the hood, we've kind of basically taken this massive Pandas API and we've narrowed it down to what we what we call an algebra. It's a subset of about 12 operators that represents all 240 operators, plus the utilities, and we basically translate Pandas into that subset and optimize for that subset. This also lets us kind of develop new APIs for data frames that might – There are a lot of like interesting, and some might say weird ways of interacting with a Pandas data frames that will be difficult to scale.

Exposing a new API that kind of narrows down the search space for the user, I think that's definitely something that we're looking at in the future, and that will be something that we can basically just reuse this optimized backend that we've been developing without having to have some of the pitfalls that Pandas has had with the tight integration between the API and the physical implementation.

[00:41:10] JM: Can you give me an example of a Mouton operation and operation? Operation on a Modin data frame and how that would get translated into lower-level operations?

[00:41:23] DP: So if we use the Pandas API, for example, we can start with something that's relatively simple. Let's start with a fillna. Fillna basically converts your null values to the value chosen by the user. Often, it's zero in numeric computation. A fillna is effectively a map where is the map function is if the value is null, then convert it or change it to zero.

And so, in the Modin algebra that we expose, this translation would be effectively a map. More complex operations like the group bys and the joins, those are also exposed. So if you want to do a group by by some key and then account, in Pandas semantics, it's slightly different than it is in SQL. In Pandas, it's a per column group by. So underneath the hood, we would do a per column group by and then a reduction on this key with account.

So a lot of these operations that you find in Spark are common in MapReduce settings. They are directly translatable to MapReduce. Then there are operations like mask, for example. What we call mask, or indexing in to the data frame arbitrarily. So in Pandas, you can kind of create a long list of indices and then just pass it to iLok, and these are the integer indices of the rows, the row numbers. The way that we translate that under the hood is with a highly optimized mask function, which basically checks the metadata, knows where each of these rows live, and then re-shuffles data if it's necessary.

The interesting part iLok and the challenge with iLok and a lot of these operators is that you can use them to physically reshuffle the data in pandas. And so one of the things that we're exploring is can we – Like, is this kind of decoupling of the physical order from the user's view? And there are a lot of challenges with this, because the user wants to see the order in the same order that it was in before. They don't want you to arbitrarily reorder your rows. And so, there are a lot of really interesting challenges that we're currently exploring on this decoupling of the physical from the logical.

[00:44:19] JM: With Modin, the Pandas API is rewritten to the Modin API before querying the Modin data frames. Can you just talk more about how the selection of that subset for the Modin API, how did you pick what operations you would put in that Modin API?

[00:44:48] DP: The Modin API you're talking about is the query compiler, I believe. And the query compiler under the hood is – The way that we chose these operators is based on what other data frame systems might need control of.

[00:45:05] JM: I guess maybe we should clarify a little bit. I was thinking about the – So my understanding is that the Modin data frames, you have an API for querying the Modin data frames, and that is a subset of the Pandas data frame operations and it gets translated down into lower-level Pandas operations. Is that right?

[00:45:27] DP: Yes. There are a couple of layers. There is one layer in between that basically to – Are you looking at the documentation?

[00:45:34] JM: No. Not right now.

[00:45:35] DP: Okay. So you're right. There is one extra abstraction layer in between the API and the Modin data frame implementation to allow other distributed data frames to implement that interface and get all of the benefits of the Modin API abstraction at the outermost layer. So, effectively, Modin is laid out in – We can consider it as four main layers, and it's a few more than that. But at a high level, four main layers. There is the API layer, which I mentioned earlier. There's a query compiler, which is where the API layer is directly translated into.

This query compiler is not tied to Modin data frames itself. It is abstracted out so that new data frame implementations can basically implement this interface and get all of the nice benefits that we have with this outermost abstraction of the API. That was kind of a community. Like we have – Some are already trying to implement this interface, and it was kind of an idea that not every not new system needs a concrete API. And we don't need distinct APIs for every new implementation. You can implement this interface and we'll just kind of have this unified API layer that all of these data frames have.

The Modin data frame is our internal implementation, and that is where the algebra comes into play. So the query compiler is a smaller subset, but it's not the 12 operators. That is something that we have worked with other groups on to see what would an outside data frame implementation want control over? What kind of parameters do they need to know about what control do they need to have over their implementation to be able to treat that system as a black box that we can just basically query into?
So that obstruction, it exists for that purpose.

The Modin layer is effectively an implementation of this kind of minimal subset, the 12 operators, and then below that is execution. And we've abstracted out execution as well, because interesting observation that I had was the data frame users, typically, they don't care about what system. Most often, it is the people who have set up their infrastructure that care. So there is an existing infrastructure that a user might have access. Perhaps, for example, it's a Ray cluster. Their company may have set up a Ray cluster, and the user just basically has only access to that Ray cluster and maybe nothing else.

In kind of the way people have been doing things, these were all siloed ecosystems and they didn't really talk to each other. And I couldn't import the same library to use across different clusters, right? Modin is aimed at changing that and making it so that your notebook doesn't change just because your environment changes. Your notebook doesn't change just because yesterday your company had a Ray cluster and today it has a Dask cluster. All of that is abstacted out so that I as a user don't really care where my code is running. I just care that I get the expected output and that it runs well and that it's fast, effectively.

So the execution environment there is abstracted out as well. And there's only about – We support Ray and Dask right now. There are only about 600 lines of code difference between them. And the Modin project itself is about 40,000 lines or so. Maybe a little bit more now. But the 600 lines difference basically gives users the power to move from a Ray cluster to a Dask cluster and back, and even back to their single node or their laptop, and that there is nothing about their notebook that has fundamentally changed. They're still importing the same library. They're still running the same commands in the exact same way. And the output is still the same. It's this idea that we expose the right abstraction to the user. And then under the hood, we basically manage everything. It's the same concept that databases have had for decades, but data frames have kind of lacked this nice unifying layer of abstractions that gives users this power.

[00:50:00] JM: So what is the kind of scale where you would need Modin?

[00:50:08] DP: Yeah, good question. So we have users who were only using it on their laptops actually for some larger datasets that Pandas is a little bit slow on. Modin also has users who are using it on smaller clusters on a daily basis. We're working with some users to kind of expand this up to a thousand node clusters and things like this. But the scale, the idea of Modin is to be efficient at all scales, and that requires some implementation. It's a challenge. It's a really big challenge to make things efficient both on the single node and in a cluster.

Spark, if you look at Spark, Spark's single node performance is often much poorer than Pandas itself. And that's just because there are a lot of overheads that come with distributed computing. You need to manage communication. You need to manage where data is. You need to shuffle data between nodes. So there are some network limitations that come into play. All of these things are challenges that we're tackling in this context of the data frame, and the new data frame algebra, and the new data frame data model and scaling this – Almost scaling the feeling that you have, this really tight control over your data and that you are the one who is controlling the data. So, to answer your question. We have users on single mode and multimode, and it's going to continue growing, I think, the cluster usage and that kind of things.

[00:51:42] JM: How do these data frames get partition in Modin?

[00:51:49] DP: Modin is designed to be as flexible as data frames are. This is a really good question, because partitioning is something that is particularly challenging in lot of existing implementations. So in Modin, we have taken the approach that the partitioning should be as flexible as the data frame is. The partitioning should be able to change in the middle of an operation, and partitions should be able to fuse together, if need be. So Modin's partitioning is such that – We can effectively treat Modin's partitioning as column partition, row partition or block partition. And that gives us a lot of really, really nice capabilities, because we can start to use optimizations for systems that are optimized for any of these. And we don't need to reinvent the wheel when it comes to optimizing for a specific partitioning layout, because Modin can be any of these partitioning layouts. And so when you look at like data skew issues, right? With this flexible partitioning schema, we can start to do work stealing, where a worker that is idle can basically steal work from a worker that is completely occupied and not available. So the partitioning is set up to basically support the full Pandas API in the first place, and also to be as flexible as data frames are.

[00:53:22] JM: Got it. So we're nearing the end of our time, and we've explored Modin in some detail. I'd like now to zoom out a little bit and just get your perspective on other gaps in tooling in the data science ecosystem.

[00:53:37] DP: Yeah. One of the things that I think is really missing is a nice set of standards, a set of standards for data frames and arrays. The challenge here is that implementations and users are interacting with different APIs all the time. If you switched libraries because library B is faster than library A was, then all of a sudden you also have to learn a new API.

So the biggest thing that I think is missing is kind of a set of APIs abstractions that allows us to not be tied to any one implementation per API. Really, what I think that we need is a good set of standards for both arrays, tensors, and also data frames.

[00:54:31] JM: Any other gaps in the tooling that you envision?

[00:54:34] DP: A lot of my feelings about gaps come from the perspective of the data scientists. The gaps in tooling definitely come from the idea that I can just scale my existing workflow, right? We've started with Pandas, but Modin eventually will be much, much more than pandas. And if we look at the way that people have been doing things in the past, there's been a lot of lock-in. There's been a lot of lock-in with APIs from the user's perspective, and there's been a lot of lock-in with environments.

I think part of this is because the people who are choosing what a company does aren't necessarily the people who are using it. And so, kind of these nice inter-plays between system, so if we look at something like Apache Arrow. Apache arrow is a really, really great step toward this cross-language, cross-system communication with zero copy. We need more efforts like this at different levels in the stack to be able to allow these levers to interplay with each other.

And from the data frame user's perspective, from the data scientist's perspective, data scientists really just want to stop using different tools to do the same thing at different scales. And so this this doesn't just stop with data frames actually. This extends to every single other type of tool arrays and machine learning and all of the tooling that the data scientists are using.

[00:56:14] JM: Devin, thank you for coming on the show. It's been wonderful talking to you.

[00:56:18] DP: Thank you so much for having me.

[END OF INTERVIEW]

[00:56:28] JM: Join GitLab on August 26th, 2020 for GitLab Virtual Commit. It's an immersive 24-hour day of practical DevOps strategies shared by developers, operations professionals, engineers, managers and leaders. The attendees will hear from US Airforce and Army, the GNOME Foundation, State Farm, Northwestern Mutual, Google, more companies. Many more companies about problem solved, cultures changed, release times that have been reduced. You can come and be part of the community that is just as passionate as you are about DevOps. You can register today at softwareengineeringdaily.com/gitlabcommit. Register for GitLab Virtual Commit by going to softwareengineeringdaily.com/gitlabcommit.

Thank for listening, and thank you to GitLab.

[END]