EPISODE 1439

[INTRODUCTION]

[00:00:00] JM: DuckDB is a relational database management system with no external dependencies and a simple system for deployment and integration into build processes. It enables complex queries in SQL with a large function library and provides transactional guarantees through multiversion concurrency control.

Hannes Muhleisen works on DuckDB and joins the show to talk about query engines and OLAP systems. If you're interested in sponsoring Software Engineering Daily, reach out to us at sponsor@softwareengineeringdaily.com. We'd love to hear from you and bring your message to our audience. We reach over 250,000 developers per month.

[INTERVIEW]

[00:00:38] JM: Hannes, welcome to the show.

[00:00:40] HM: Thank you for having me.

[00:00:42] JM: You are working on DuckDB, which is an approach to doing OLAP processing. Can you give an overview of the current state of the art different options for OLAP query processing? And maybe give an overview for what OLAP is for people who don't know.

[00:01:01] HM: Sure. Happy to. So this comes a bit from the SQL world. And there's this general misconception of people that kind of connect the language to specific systems. But there is a whole – I mean, many people notice. But some people don't. So I tend to say it. There's this whole wide range of different implementations of the language if you want execution engines for the SQL language that exists out there. And there's like two main, let's say, classes of systems that exist there. And one is the OLTP, online transactional processing, and one is OLAP, is online analytical processing. And these are very, very different in the kind of use case they're used for. So most people are familiar with OLTP, because the systems that people are very familiar with like MySQL, Postgres, SQLite, traditional Oracle SQL Server, the whole list. They're all OLTP, or at least they come from this world.

And that's great if you're running like a web shop or something like that, because your queries will be focusing on individual rows. You're adding rows. You're deleting rows. You're updating rows. This kind of stuff. But once you're trying to do like analytics, like driving a dashboard, for example, creating reports, statistics, ML, what have you, these systems are not really ideal anymore. And people have been designing custom SQL systems called OLAP systems for this purpose. And I think these are less sort of well-known. Especially in the open source space, there's not a lot of them.

Traditionally, I think for a long time, there was something called MonetDB, which was kind of the only representative there. There were some commercial systems. And these days, I think what people would be most familiar with is ClickHouse. Maybe you're aware of it.

[00:02:52] JM: Sure. Yeah.

[00:02:53] HM: Yeah. So ClickHouse, I think, is at the moment the most well-known system there. And then of course you also have the distributed camp, like Spark, and Presto and all the other systems in that class. But I think, yeah, this isn't entirely kind of – That's another camp yet, the distributed big data SQL kind of systems. But I think ClickHouse would be the most well-known representative there. But we don't work on ClickHouse. We work on DuckDB, which is I don't think yet another, I think, take on this OLAP problem.

[00:03:26] JM: These different systems have tradeoffs in terms of latency, cost, complexity, API. You are working on DuckDB. Can you give an overview of how DuckDB compares to these different OLAP systems and what the most important competitive differentiation is?

[00:03:53] HM: Yeah, happy to. So I think what the design goals for DuckDB, they came kind of out from talking to people, data scientists specifically. We're not using databases. And we try to figure out why that was. And they gave us all sorts of reasons. For example, they said, "Oh, these databases are horrible to use. They're difficult to set up. They're difficult to maintain. They're difficult to get running in the first place. Have a lot of problems with data transfer in and out." So this kind of drove the design goals for DuckDB. So we try to differentiate by being super easy to install, which is why DuckDB is a zero-dependency 10 megabyte kind of binary that you install without like anywhere really that you can just run.

And I think the biggest difference is from the kind of architecture level where traditional systems, I think, without exceptions, are these days all client server, where database server sits somewhere and you have a client protocol, a client program somewhere maybe as a library somewhere. And between those client libraries and the database server you have some sort of client protocol. DuckDB is very, very different there because it's a database as a library. We call it the in-process OLAP. And it's the first one, because we kind of invented this as a class of systems, the in-process OLAP that just doesn't didn't exist before. And it's kind of interesting. Sometimes we're wondering why nobody has kind of done this. But there we are.

[00:05:26] JM: So when you say it's an in-process SQL OLAP database management system, what does in-process mean?

[00:05:34] HM: Yes. It's is super cool. You're aware of SQLite, right?

[00:05:38] JM: Yes.

[00:05:38] HM: Okay. So SQLite doesn't have a server to speak in that sense, right? The way you interact with SQLite is you link that to your application with like a shared library. Or you even compile it in. It's also possible. And then there is no kind of separate process even that runs your database. The database is just a set of API calls into the SQLite library. And DuckDB works the same way. So that, yeah, basically the way you run it is it's not running as a separate process. You will be basically starting up the database.

So within your application, for example, very often that application will be Python. And then your database server, in this case, DuckDB, will run inside the same process. And yeah, you then can run queries there. And there are threads running and all the stuff that normally the database server does is happening in the process you're in. And it has one crucial advantage, and that is that your database runs where the data is. So, say, you have data sitting in Python or you have data sitting in your client application, if you run the database server inside the same process, you have a giant advantages in terms of data interchange.

So, for example, very often, we can actually do query processing without actually copying the data into the database, because sitting there in-memory, it's just bytes. DuckDB knows how to read a wide range of different data formats from memory. And then we can just run queries on that directly. And that has – Yeah, and it's especially relevant for these analytics use cases where you tend to transfer large chunks of data to your application. For example, if you want to train a model or you want to do some complex visualization. This is something that we found that is less relevant in the transactional use cases where, typically, you don't push millions of records to the client. But in the OLAP cases, that really happens. And yeah, traditional systems are not very good at that. We wrote a paper about that. So I think that's the main difference.

[00:07:39] JM: Gotcha. So can you give me a description of the database architecture, like the backing storage system is? And then we can go into query engine and so on.

[00:07:54] HM: Okay. Happy to. I mean, SQL systems, they have this interesting thing that the interface is pretty clear, right? You put in a SQL query and you expect some sort of result. It's pretty straightforward. Simplifies your system design. You don't have to think about it.

And maybe we can kind of walk through there like a query would walk through there, right? So we start with the parser obviously. And what we have done there, actually we're using a project that somebody has made that wrapped the Postgres parser in a separate library without the rest of Postgres. And we've heavily patched that thing again. But basically, we run a derivative of the Postgres parser for transforming the SQL text into a parse tree. And that has a one big advantage that we are quite compatible with Postgres, is query syntax, right? So that helps, because lots of people know that particular SQL syntax.

So then we have a parse tree. We take that. We transform it to a query plan, a logical query plan. That's our internal representation. We have an optimizer that optimizes the query plan. And it's another version of the logical query plan. And then that goes into a pipeline-based vectorized execution engine, which is something that – Vectorized engines are, let's say, known, but they're not incredibly well-know. This is a technology that is particularly useful for OLAP kind of queries.

Yeah. And then on the storage side, we have full MVCC transactions on top of a storage format that is a single file storage format, It's actually quite – Basically, the entire database lives in a single file. And there is just like fixed size blocks that contain the table data in a vectorized columnar sort of data representation where you have columnar data representation, but you chunk these columns up into row groups to, yeah, localize rows a little bit.

What is important, though, is that DuckDB is different from many other systems, and that it doesn't have to actually own the data to run queries on top. I've briefly referred to this before. But basically, if the data is kind of compatible with what we understand, we don't have to actually import anything to run queries. For example, we can run SQL directly on Pandas data frames, our data frames, streams, all sorts of stuff. And this is kind of because we think it's cool, and it's also because I think there has been, from our experience with talking to people, a bit of resistance of handing over your data to some other people's systems. Because the people have made this experience that once they do, they never get it out again. So we say, "Okay, maybe we can just run queries on the data wherever it lives. But this is kind of the structure. I'm happy to go into detail on any of these things. Maybe, for example, the vectorized engine that's a bit, let's say, unusual. Or anything else. What do you like?

[00:11:01] JM: Let's go through the step-by-step read path of an OLAP query in DuckDB.

[00:11:09] HM: Okay. So when you're reading from a table, is that a –

[00:11:13] JM: Yes. Absolutely.

[00:11:15] HM: So when you have a select query, right? That would be the way to read stuff in SQL. And that becomes a logical operator for the scan. And as I mentioned, the scan can run from multiple sources. We'll just assume that we read from the internal storage for now, which means that it's actually interesting. Maybe I can mention this briefly. Because we run on so many different data formats, we actually treat all the different scans the same way, including our own scan. So inductively, all the scan operators are kind of pluggable. And you can have your own. You can kind of bring your own scan operator, if you want. And some people do.

But let's just assume we use our own storage. So then there is this – On this cloud format has these fixed-size blocks to store the tables. And basically, in those blocks, there's table data in these row groups that contain columns for the individual columns. And the scan operator will start reading vectors of data from these blocks. And that means that it will – Traditional systems, they read the data row by row. They'll not do this. We also will not read an entire column of the data like some other systems. We'll read a row group, which is based in our – The default is I think a thousand rows or so. But we will read column chunks of a thousand values for the entire table, or at least the part of the table that we need for the query. And then we have something we call beta chunk. And that becomes the intermediate as an input for the operator that sits on top of the scan, and can't be really arbitrary. It depends on what the query wants to do. It could be a filter. It could be an aggregation. It could be a join. It could be a sort. You have it.

And basically, the important thing is that the basic unit of data that flows through DuckDB execution engine is these vectors of, as I said, by default, 1024 values. And that has – There's a reason for this. And the reason is that the vectorized engine – It's called a vectorized engine because it uses these vectors of data that does this, because it doesn't actually compile queries like some other engines do this. But it still interprets the query plan. But because it interprets the query plan, there's a slight overhead in the interpretation. And to kind of amortize this overhead, we just look at a bunch of values at the same time.

Another big advantage of using these vectors is that if you have an expression – For example, if you say like A plus B plus C in your query, in the projection, because we are always operating on these fairly small vectors of a thousand values or so, these intermediates of these computations, they stay in the CPU cache. And that is a very good thing for performance. And that's kind of the design rationale behind this vector engine.

And basically, yeah, the entire engine operates on these streams of vectors, right? So you have kind of streams of sets of vectors. And every single operator in the engine basically deals with this format. Of course, there's a bunch of operators that we call blocking operators that have to materialize the input. For example, in a join, you have to materialize one side of the join. Or if you want to sort data, you have to materialize.

But apart from those, it's basically the engine streams these sets of vectors through the query plan. Does this automatically in parallel. It's also pretty cool. So we take the query. And we have this pipeline parallelism engine that basically automatically figures out how to run the query in parallel. And we will use all the threads that are available. So that's, I think, roughly how that works.

And once we are done kind of, or once we have kind of pushed this stream of vectors to the entire query, you get it as a user. So basically, you will be able to – The client API will deliver those vectors to you if you want. And because you're in the same process space, you can basically just directly interpret these. And that's all fairly efficient. The main metric for OLAP engines is something we call cycles per value. Basically, that says how many CPU cycles are we spending on processing a single value, a single row, a single value, whatever you like, of the input? And that's really the biggest sort of target metric that we're trying to optimize for, because it's really super relevant. And there is easily a thousand X difference between SQL systems in this particular metric.

[00:15:57] JM: So it's worth talking here about other alternative query methods. So if I was to use Snowflake, for example, for issuing a large-scale query, how would that execution path differ from that of DuckDB?

[00:16:18] HM: That's interesting, because one of the co-founders of Snowflake comes from the same lab **[inaudible 00:16:23]** where DuckDB comes from. So Snowflake, as far as I'm aware, also uses a vectorized execution engine. So it would actually be fairly similar, I think, in sort of the big lines, the general scheme of things.

I think what you have in the Snowflake case is that you have the distributed execution path, which DuckDB is focusing on the single node, which is a pretty huge architectural difference, let's say. And I think this is kind of interesting because, people, for the last, I don't know, 20 or so years, have been working really, really hard on building these distributed execution engines. And in my opinion, people have been kind of neglecting the single node execution performance a lot. As a matter of fact, they've been neglecting it so much that single node execution has actually become slower. I think people have also kind of forgotten how much you can actually achieve on single node. So I don't know. I have this MacBook here. And maybe have you heard of this taxi data set, the New York City taxi data set?

[00:17:25] JM: Sure. Yeah.

[00:17:26] HM: Yeah. Usually, this is used a lot in benchmarks for big data systems, right? You can basically run queries on that thing in its entirety on my MacBook here with DuckDB and not use a distributed system. And I feel like this option has kind of faded into obscurity for some reason, even though it is for like the overwhelming majority of datasets. The much better option than going for distributed.

I think there's – I think if you wonder about the difference, let's say, between how DuckDB does with queries, versus what Snowflake does with queries, I think something like Snowflake or Presto will spend a huge amount of time on the actual coordination of the distributed query execution including, for example, partitioning and shuffling stuff around coordinating execution, pulling results back in. And they're spending a lot of time on that. And I think in many, many cases, that actually will dominate the execution time. So if you're not doing that because you have a giant computer with a terabyte of main memory or something like that, you can gain a lot, a lot of performance.

[00:18:39] JM: And when you say performance in this context, do you mean speed, or cost savings, or what?

[00:18:44] HM: I think it's both, right? I think what has been kind of recognized at this point is that the cloud SQL systems are not really able to deliver interactive performance, right? So if, for example, you have something sitting in Presto and you're clicking a button, if whatever the button needs to do requires a big query running in Presto, you will not be able to do this interactively, really. This has – I think, it's been promised a couple of times. I don't think it really has happened.

Same with Spark. They talk a lot about how they use memory. But I haven't seen a lot of Spark jobs completing under a second, let's say. So you can get a lot of quick performance from using something like DuckDB, which just doesn't have to do this all coordination game with hundreds of nodes. And of course, if you have higher single node efficiency, you will also spend less on your cloud provider. This is one experiment. It's been a while that we ran it. But we run a query on this taxi data set, used DuckDB. And then we looked how big does the Spark cluster have to be to match this performance, right?

So take one node, runs DuckDB with this particular query. Okay. Now we have to see how many of those nodes do we need to beat the square performance with Spark. And I think the answer was 32 or something like that. And that's a 32X increase in – I mean, you have to pay Amazon 30 times the money for that. So that's pretty significant, I would say.

And in general, as I said, there's this misconception that everybody is sitting on these petabytes. But while some people do, and I have spoken to some of them, I would think – Also, studies show that the majority of data sets people actually look at easily fits on, yeah, something like a laptop. And it's more like that there has been a bit of a sort of a shortage of tools that can actually take advantage of something like a laptop. Because what can you really do if you want to do something like OLAP or general data wrangling of relational data on something like a laptop? What are your options there? You have Pandas or something like that, right? Could potentially install ClickHouse. But it's not like there's a ton of great options to get really good processing performance on the computers that you have, right?

[00:21:15] JM: Yeah, I suppose so. The one question I have about DuckDB is around data management in terms of memory usage. So if I'm performing a large query or a join, I'm going to be pulling a lot of data into memory and performing operations on top of that. And depending on what programming language I'm executing that in, there might be different data structures that might be pulled into memory. And I know you use Apache Arrow to have data integration between Python and other data formats. Can you just talk about memory management in more detail and how apache arrow fits into that?

[00:22:11] HM: Certainly. Yeah. So use Arrow as an integration platform. So, yeah, indeed. But we only really use it at the boundary. So we can read Arrow data as if it were a table in DuckDB.

And we can export query results back to Arrow. And this both can work with zero copy. That's not so relevant for the internal state of the query engine, right? Because we're not using Arrow there. But we use a similar, but it's not the same.

Let's say we do a join. For memory management in a join, obviously the join is one of these blocking operators. If you, for example, run a hash join, you have to materialize. You have to build a hash table on ideally the smaller side of your join, and that needs to sit somewhere.

Now, traditional in-memory engines would just put that hash table in-memory and would essentially, yeah, build it in memory and then scan it in the probe phase. That works. And this is also what DuckDB does by default. I mean, if you have enough memory, we will just use memory. We will put a hash table that is a custom hash tables binary encoding where we – It's kind of funny, because hash tables are one of these places where columnar engine is kind of forced to encode data in a row major format again because it's just the way how you're referring to it in a hash table, right? So you kind of have to give up on your columnar dreams in hash tables also, in aggregate hash tables, sorting and so on and so forth. Kind of funny. But then this hash table sits in memory.

Now what happens if this becomes too big? And this is super interesting, because it's something that we're spending a lot of time on at the moment and on working on this thing. And we call the never give up, never surrender project. It's because we kind of want to make sure that even if you're running out of memory, we still are able to complete the query successfully.

And so this is where we actively work on. And the general idea of the general philosophy that we're following is to kind of gracefully degrade as we start using disk. This is different from what traditionally happens. Because in traditional systems, what will happen is they will run happily in-memory until they've realized, "Oh, this doesn't fit in memory anymore." And then they switch to an entirely different algorithm, for example. Go from a hash join to a stored merge join. And in DuckDB we've decided that we don't want to have these hard cuts in operator implementation because it creates these unexpected performance clips. And it's something that we've learned that people really dislike about relational systems is that there are these – Your query is the same, but you add one more row of data and suddenly an entirely different algorithm is executed and they have no kind of handle on this.

So what we like to achieve there is this graceful degradation of performance as we run out of memory by using the disk to basically write stuff to this that we can kind of do without for the moment while still achieving progress. And the way we can do this – For example, I’ve spent a lot of time on the hash aggregate. The way you can do this is by partitioning the things like hash tables, for example, by radix values. And then basically dumping stuff to disk that you're currently not needing to match hash values, things like that. So there is some active, actually, research in our group currently on making this out of memory situation kind of survivable, which I think is kind of – Yeah, it's kind of new for an in-memory OLAP engine. Because as I said, traditionally, they have just assumed that the working set that is the data that query actually requires to complete will somehow fit in memory. And we are kind of working on changing that. It's also because DuckDB is aimed to run kind of anywhere. And by anywhere, we also just – Devices, like phones or something like that, that have a different kind of memory set up than, let's say, your high-end server.

[00:26:21] JM: What is the typical deployment model for DuckDB? Like, what are the the memory and storage requirements, and like processing?

[00:26:32] HM: There's really not a lot of requirements. It works really well on Raspberry Pi's, if you want. In fact, we've had another experiment. We run DuckDB on a Raspberry Pi, and we run SQLite on a $5,000 server, running the same benchmark. And DuckDB ended up being faster on the Raspberry Pi. Of course, that was an unfair comparison because we were using an OLAP benchmark which SQLite is not built for. But it was interesting to see. Our general goal is really that it runs wherever. It should run from anything. Single core, 500 megabytes of RAM, up to a server that has 200 cores and a terabyte of RAM. And in fact, we have one of those, and it works perfectly well there as well.

I think in terms of typical deployment, we see a lot of MacBooks, because DuckDB is used a lot by data scientists working from R and Python. And yeah, it's just people on their laptops basically. And that's – I don't know. What is that these days? Maybe eight gigabytes of RAM? Something like that? And as I said, it's quite impressive what you can achieve with that kind of hardware if actually using it. And it's kind of interesting because it's really quite rare in these data science tools to, for example, parallelize things. It's only now starting to become a thing.

And by kind of parallelizing, auto-parallelizing queries, we can kind of use that hardware to a much larger degree. There have been some issues reported where people said, "My computer suddenly becomes much hotter than it has ever been." But that's kind of – That's I think something that we have to live with.

[00:28:16] JM: You wrote a blog post about using WebAssembly together with DuckDB. I guess this wasn't written by you specifically, but this is written by your team. And given that this is what I would consider generally a back-end technology, an analytic database, it's interesting to see a use case of browser-based analytical query engines. Why would it make sense to use to use an analytical database in the browser? And why would WebAssembly be a good application for that?

[00:28:56] HM: This is one of these things that have blown my mind completely. So this was not initiated by us directly, but by Andre Kohn, who is from the Technische University of Munich. And we work together now. The reason this – So what he's done is to basically take DuckDB and cross-compile to WebAssembly. The reason he can do that is because DuckDB doesn't have a lot of dependencies, right? So it's kind of lightweight in terms of that. So this is why it was possible to cross-compile to WebAssembly and then run the entirety of DuckDB in the browser.

In fact, we have a demo on our website where you can basically just run DuckDB in the browser and get a shell and you can type queries there. It's kind of funny. Why would you want to do this? I think Andre might be the better person to answer this question. But in general, we see a lot of data being pushed into the browser. And it has to do with the backend latency.

Again, if you're trying to build an interactive visualization and you want to have something like a slider, something pretty trivial, if every slider move requires a call to a backend and possibly a Snowflake query or of a Databricks query, that's not going to be happy eyeballs. It's just because the backend latencies are too large. So it doesn't make total sense to push a data set into the browser directly and basically recompute your visualization input there instantly.

And it's really funny if you ever see this in a demo. It's so instantaneous that you wouldn't believe it. So I think the reason why people push OLAP engines into browsers because of latency and because of these vastly and kind of under-appreciated capabilities that you have in browsers now with this WebAssembly technology that allow you, yeah, to run something like an analytical SQL engine entirely inside the browser.

And what we, for example, also see that people take their big and backend database system and then they export data that is relevant to a particular visualization as something like, for example, Parquet files, which DuckDB can read natively. And then also from the browser, by the way. And then they export this Parquet file into the browser and have the visualization generate their own SQL queries and run them on this subset of the data that if they export it for this particular purpose. So that's I think the reason why people run this in the browser. We're also working together with a company called Stoic that is basically running a data analysis workbench as a progressive web app, or part of it at least, which is it also uses DuckDB.

[00:31:47] JM: And can you speak at all to the functionality of WebAssembly and in allowing that browser-based OLAP querying?

[00:31:58] HM: Yeah. I mean, what do you mean by functionality?

[00:32:01] JM: So why do you need WebAssembly? I guess you just need webassembly because that's necessary to run C++.

[00:32:07] HM: That's right. I mean, you could of course write a SQL engine in JavaScript. And some people have tried. But I think this – Yeah, I'm not entirely sure whether this would be really that fast. In WebAssembly, yeah, basically you can leverage existing implementations in C++. And since – I mean, DuckDB is implemented in C++. So you get kind of the WebAssembly version as a side product, if you want, right? Because you can "just", because there's some work required to make this work, of course. But you can take an existing engine, in this case, DuckDB, and just basically cross-compile it to WebAssembly. And then you can run it in browsers.

I think the capability there – The surprising capability, in my opinion, is that this is something that has been built into browsers in the first place. So my kind of 90s kid sort of brain is still slightly warped by seeing what kind of tech people push into browsers. But that's a different discussion obviously.

Now, I think it's really cool that you can cross-compile projects like DuckDB and just run them in the browser. And we've done some benchmarks to see what the performance hit is if you do that compared to the natively-compiled C++ version. And it wasn't so crazy. It was something like 30% slower, which really isn't so bad if you think about it, if you consider what kind of machinery is running there to make this work.

Yeah. And I think it's really just super exciting to – We were super excited to see just this work because, yeah, it opens an entirely new kind of world where traditional OLAP engines would have never been able to set foot into kind of. Because, yeah, as I said, the traditional engines, they are far too heavyweight to be considered for these cross-compilation games. You have hundreds of dependencies.

I don't know if you ever tried to install something like ClickHouse. But I did. And we were looking into hundreds of dependencies, including like three compilers or something like that. And that's just not going to happen to be able – You're not going to be able to put that in the browser. But that's actually one of the advantages of vectorized execution engines compared to the, let's say, more standard just-in-time compilation engines, is that they can be built much more lightweight because you don't need the entire compiler infrastructure, because it's still an interpreted engine. It's just more efficient than your normal interpreted engines.

[00:34:46] JM: Could you dive a little bit deeper into that when you're comparing the – You said the interpreted engines versus the compiled engines.

[00:34:56] HM: Yeah. There's an interesting paper for people that are interested. I think the title is something like *What You Always Wanted About Vectorized Versus JIT But Never Dare to Ask*, or something like this. So I mentioned before that the metric that we are looking for here is cycles per value, right? So like we want to build an engine that runs queries that is supposed to be as efficient as possible with regards to how many CPU cycles are spent per CPU value.

Now, the reason that traditional systems like Postgres are slow or spend a lot of cycles on each value is that, for every row of data, they have a lot of levels of indirection, function point, just these kind of things. A lot of switching between types for every single row that makes it basically pretty slow to process queries in general. That gives them a pretty large cycles per value.

So basically, these traditional engines, they use a lot of cycles per value because of the interaction per row in directions in their engines. Now, how can we get rid of these? Well, one way is to JIT it. Basically, you say, "Instead of interpreting my query plan on a row-by-row basis, I will turn my query plan into an executable program by JIT compiling the query into a program." And that's what engines – Like, this has be done all the way back in system R when relational engines were born. And it's more recently done in OLAP engines by a system called Hyper, which is now part of Tableau. Also comes from the big university in Munich.

The thing is, though, with these JIT engines, there's a quite high complexity in building them, because I have to imagine that you're generating code. And if something goes wrong, you have to debug generated code, which is, I'm told, pretty gnarly. Also, as I said, you need a lot of compiler infrastructure to turn a query plan into an executable, because this needs to be done on an assembly level, right? Or an LLVM assembly level, kind of the same thing.

Now, a vectorized engine, as I've mentioned before, tries to amortize the overhead of interpretation by just making every individual operator that it contains, the engine contains, able to deal with multiple values at the same time. So instead of operating on single values at a time, DuckDB – For example, the addition operator will be able to take two times 1024 values and produce 1024 new values as a result.

And that way, we can also get the cycles per value down because we're able to basically only have to do the switching around on types and which operator we have to run once for every 1024 values at a time. And that really drives down the cycles per value. And that is really what makes it go fast.

And I think that's – And the big advantage, I think, of vectorized engines, and there's an ongoing discussion in the academic world on which one is better. My take is they're generally comparable. But they have advantages and disadvantages. And one of the advantages of vectorized engines, that they can be built much more lightweight. And it's I think one of the reasons why DuckDB was able to stay fairly nimble despite us supporting a huge chunk of the SQL standard.

[00:38:28] JM: What's your strategy around monetizing DuckDB?

[00:38:33] HM: So DuckDB was built first at the CWI. I don't know if you're aware. The CWO is the Dutch national research institute for computer science and mathematics. It's where **[inaudible 00:38:42]** was a long time ago. It's where Guido van Rossum invented Python not so long ago. And in that research lab, there's a database group, and I'm part of the database group. And we have kind of built DuckDB in the first place there because we thought it was the right thing to do. And we started building it like four years ago. We got some grants to work on it. But it's also – It gets a bit tricky at some point to kind of build a software development team inside a research institute, because a research institute is not necessarily meant to do this, right? They're meant to write papers and educate PhD students and such things. But kind of building a software development team inside a research institute has its limits, let's say.

So last year, we spun out DuckDB Labs from CWI, which is a new company that is now working on pushing DuckDB further. And I'm spending most of my time in DuckDB labs these days. I'm also the co-founder and CEO.

And how do we monetize this? Well, we haven't taken we VC money. So we're not sitting in a giant pile of cash that we're slowly shoveling into a big fire. But we are actually using this very traditional method of having customers that pay us for our services for services around DuckDB.

So, for example, DuckDB of course is free and open source. And most of our users don't pay us the vast majority. Our users don't pay anything. And we're very happy about that. But there are organizations that want to use DuckDB as a building block in their infrastructure, right? So they want to build like a large data pipeline thing around DuckDB, for example. And then maybe this happens, they have some sort of custom requests, or they want us to prioritize work on specific issues. And so the way DuckDB Labs makes money is by contracting with people or companies that need kind of custom work around DuckDB, or that need specific things to be prioritized. And we charge them for that. So that's kind of how we do that.

At the same time, we also collect donations in a separate entity called DuckDB Foundation. Basically saying, "Okay, if you would like to support the development of DuckDB, you can become a supporting member of this foundation," and we will use that money to pay for the maintenance. So that's kind of the model that we have.

[00:41:19] JM: And what are the types of users that you're seeing adopt DuckDB?

[00:41:28] HM: Yeah. So there's, I think, two camps, I would say. There's the data scientists that are just using DuckDB on their on their laptop. And there's quite a lot of them. There's quite a lot of them. But they are not necessarily – They just tend to be happy with the way the thing is. And then there's the other camp, which I would say is other software companies in that sense that are building larger infrastructure that uses DuckDB as a building block somewhere. Basically, uses of DuckDB of the database as a library, or maybe it's also quite common that they only use some components. For example, we have some customers that are not using our execution engine but are only taking the plans as they are generated and do something else with them. Or there's yet another customer that doesn't want to do that, but wants to use the execution engine, but not the SQL frontend. So they generate plans themselves and hand them to us for execution. So there's all sorts of combinations of views. And I think those are the more complex use cases and those are also the ones that we then work with them to add hooks.

And in general, so far, it's been working out really great kind of to take these requests for changes. Think about ways in which we can make something generic that will benefit multiple people or multiple kinds of users and build that into DuckDB, and thereby pushing the platform with these requests from individuals.

So I think one thing that's also different about DuckDB is that if you're not on an Apache project or anything – And since we kind of started as a sort of new take on OLAP, it was a bit opinionated. Let's say a bit opinionated at the beginning. But we've kind of kept this opinionated design in that sense that we still think very, very hard whether a particular feature makes it or not, and whether this makes sense for the overall project or not. And I think that has so far worked out pretty well to keep the thing clean, and small, and efficient.

[00:43:34] JM: Well, let's close off with the toughest engineering problem that you've had to solve when building DuckDB.

[00:43:41] HM: Here, I can praise my co-founder, Mark Raasveldt, who is the CTO of DuckDB Labs. Was my PhD student long time ago. And there is something called subquery folding. Are you aware of this problem?

[00:43:54] JM: No.

[00:43:57] HM: I thought so. It's obscure. But in SQL, you can have a sub-query, where basically you have a query inside the query, right? Have you seen that?

[00:44:04] JM: Sure. Yeah.

[00:44:05] HM: Okay. And you can have a correlated subquery where you have, for example, a comparison, like a filter inside a subquery that refers to things from the encapsulating query using that. Yes, that's called a correlated subquery.

And the problem with correlated subqueries is that there is a naive way of evaluating them and a correct way by elevating them. The problem is that, in the naive way, we kind of have to conceptually re-evaluate the subquery for every row of the encapsulating relation, right? So for every tuple that we look at in the outer query, we have to kind of rerun the subquery with the correlation variables field from the outer relation. This is what Postgres does. Now this works for Postgres because they're an OLTP system. If you try to do this on a table that has a billion entries, you're basically re-evaluating the subquery a billion times. This is not going to happen, right?

So in order to fix this, you have to do something called subquery folding where you cleverly rewrite the query into a join with new and exciting join types to basically eliminate the correlated subquery and replace it with a strange join. And Mark managed to implement this in something like two months and then basically turned up and was like, "It's done." And I really was absolutely blown away that this was possible, because I had looked at it and was completely overwhelmed by the involved evilness, let's say, of this problem. It is something that is really,

really badly described in the literature. Nobody really does optimization. Like, nobody really talks about it.

There was one paper, again, from the University of Munich, that kind of described the path towards success. But I was absolutely impressed when that particular problem was solved. I think that's the, let's say, overall toughest problem. I'm just trying to think of my personal artist problem inductively. And there is one, but I can't think of it.

I think reading Parquet files may have been not a tough nut. So, you know Parquet files, right?

[00:46:22] JM: Yeah, certainly. It's just like a tree – It's a kind of a tree style compression.

[00:46:33] HM: Yeah, can be trees. Yeah. It's like a columnar table format for like on files on disk, right? It can be recursive, but it's usually tables. And this Parquet format is everywhere, right? It's like the standard table format. And the thing is everybody just uses an off-the-shelf reader for this. And everybody tends to use the same reader, which is Parquet – There's a Java thing called Parquet MR or something. Comes from MapReduce way back when. And for various reasons and also because we hate dependencies, we decided to write our own.

And it's one of these problems where you start digging in like a file format where there is like a reader for it, but the specification is kind of whatever that reader does, right? Or whatever the writer that does. So it's like a file format where there's not really a great documentation. But you kind of have to dig through the various implementations that exist to kind of understand what on earth is going on. And so I spent an ungodly amount of time understanding like, for example, the encoding of nested tables in parquet. But of course, the great result is that I think DuckDB now has a very competitive Parquet reader that has zero dependencies. So that's something that something good came out of that.

[00:47:54] JM: Awesome. Well, listen, thank you so much for coming on the show and talking about DuckDB. Exciting to see another entry into the world of OLAP querying.

[00:48:05] HM: Thank you so much for having me. It's been a pleasure.

[END]