Goetz Graefe Speaks Out on (Not Only) Query Optimization

Marianne Winslett and Vanessa Braganholo

Goetz Graefe
https://dblp.org/pers/hd/g/Graefe:Goetz

Welcome to ACM SIGMOD Record's series of interviews with distinguished members of the database community. I'm Marianne Winslett, and today we are at the 2017 SIGMOD and PODS conference in Chicago. I have here with me Goetz Graefe, who is the recipient of the SIGMOD Innovations Award, the SIGMOD Test of Time Award, the ICDE Distinguished Paper Award, and the ACM Software System Award, all for his work on query processing. His Ph.D. is from the University of Wisconsin − Madison. Goetz was an HP Fellow and he currently works for Google. So, Goetz, welcome!

Thank you.

You've worked at a lot of different places but always on query processing. Everybody wants to know what it's like to work on the same topic for 30 years.

I've worked at multiple places. I've taught at Oregon Graduate Institute and then at the University of Colorado at Boulder, and then at the Portland State University. Since then, I worked for 12 years at Microsoft and then for ten years at HP. So, I wouldn't say I worked at a lot of places in the last 20 years.

I also didn't always work on query processing. I worked on query processing as a graduate student, and then as a professor, and then for a while at Microsoft. But then I switched to working on indexing within the SQL Server product. And at HP, I worked on query execution again. And then, I also worked on concurrency control and on write-ahead logging and recovery. So, I've always worked within the database engine, but not exactly on one topic only.

Okay. So, why does everyone think you've been working on the same thing for 30 years? Is it because you're Mr. Query Optimization?

Actually, I don't know whether that is even true because, in Germany, I'm actually often known as Mr. B-Tree. Obviously, Rudy Bayer invented B-trees, but I have published probably more than half a dozen papers and surveys on B-Trees. So, in Germany, people think I'm Mr. B-Tree.

Okay, we're correcting that misperception then of 30 years of the same thing.

I like to believe I'm neither one. I'm neither only query processing nor only B-Trees. And I think there are a couple of pieces of research hopefully coming out soon on concurrency control, and there actually are several pieces out already on logging and recovery[1]. So, hopefully, that perception will vanish over time.

It sounds like you're a full-stack guy for the database engine itself.

Perhaps something like that.

People would like to know, are there still open problems in query optimization?

There most certainly are. Plenty of them. But they all boil down to fairly few topics. So, the way I think about query processing is No. 1; people expand the functionality of query processing. People put more and more analytic functions into query processing. People want to put all kinds of clever machine learning into the query processor. So, expanding the functionality is one thing.

The other thing (No. 2) is most people when they think query processing, they think about traditional relational query processing, and then they think about performance. And performance, I really divide into three things: (i) efficiency, meaning clever algorithms that process data fast; (ii) scalability, clever ways of using many, many computers (or at least many, many cores). And the third one (iii) is really robustness of performance, and with robustness, I mean more than predictability. If I have a car that never starts when it's wet but always starts when it's dry, it's a predictable car, but it's not a useful car. So, what I want is more than predictability. I want robustness, something I can rely on. I want the good performance every day. I'm okay if I don't get best performance. Maybe I don't get best performance ever.

And my standard analogy for that is if you own a house, every year you pay good money yet you hope you will never get anything back for it. It's called fire insurance, right? You pay it, and you pay it willingly as long as it's a small fraction of the value of the house. And as long as you can count on if you lose the house due to fire, it'll get replaced. So, then similarly with robust performance, you're probably willing to forego some small amount of efficiency, if, in return, you get robust performance, predictable performance, reliable performance, which among other things, permits you to load your service much higher.

If you get random load spikes and you never know when, you end up running a service at 20 percent utilization. But if your performance is very steady and utilization is very steady, it's perfectly reasonable to run at 60 percent utilization. And suddenly, the ten percent overhead – or maybe even the factor two overhead – comes back and more, if you can load your service substantially higher.

[1] The interested reader can refer to these publications:

Goetz Graefe: On transactional concurrency control. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2019.

Goetz Graefe, Wey Guy, Caetano Sauer: Instant recovery with write-ahead logging: page repair, system restart, media restore, and system failover, Second Edition. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2016.

That's a great lead-in for something else I wanted to ask you about. There's this recent work from Rick Snodgrass's group that suggests that there's an inherent hard limit on how well top-down rule-based optimizers can do. And the kind of behavior that they're seeing in their experiments with commercial engines is exactly what you describe as the key thing to avoid if you wanna have robust performance.

> ***[…] choice is confusion.***

It's like the system somehow ends up with too many options to consider, and it does worse and worse on an average random query. Not the ones it was tuned for, but just something that's slightly different. Do you think there is a wall there and that rule-based query optimizers have hit that wall or will hit that wall?

Well, I don't think it has anything to do with what kind of query optimizer you have. Whether it's top-down, bottom-up, rule-based, transformation-based, dynamic programming – they all have the same problem. I think the key issue here is also not how many logical operators you have like join and select. I think the bigger issue is how many physical operators you have. So, not how many algebra operations you have in your specification algebra, but in your execution algebra. And so, if you have 17 join algorithms in your system, chances are you'll hardly ever pick the optimal one. In fact, you should be happy if you always pick a good one. And it's unlikely to be the case.

So, the fewer algorithms you have – ideally, if you only have one – you can never choose a wrong one. So, yes, there is a practical limit, and I think the practical limit comes from two things. No. 1, choice is confusion. If you have too many choices, you get confused. No. 2, cardinality estimation will always be inaccurate. No matter how sophisticated your model is to describe the distribution of data values, there will always be perhaps an adversarial case – as a test case – where the model that you have chosen to implement does not capture the distribution you truly have.

And so, I don't believe that the solution for the lack of robustness in query performance will come from the planning part of query processing. I actually believe it will come from the execution part from query processing.

So, what do you mean?

Well, I think plans will often be right and good. And there will always be cases where, in particular, the compile-time planner will choose a bad plan. And what we really need is execution engines that are much more forgiving. So, the word that I choose here is graceful degradation. It's very important. The problem in products is that the customer complains about a bad plan having been chosen for a query. Now, a bad plan chosen sounds of course like a defect, a bug, a complaint that should go to the query optimization team. So, the query optimization team will do what they can to have a different plan chosen or a better plan chosen.

Maybe they make the cardinality estimation or the cost calculation more sophisticated. But I think in some sense, it's a futile battle. I think that in many cases, the solution will come from the query execution engine being more forgiving about what plan actually got handed to the query execution. So, can the query execution engine somehow avoid performance deterioration that is not graceful? And can the query execution engine execute the plan in a way that doesn't show the mistake as badly as a naïve query execution engine would?

So, what exactly should be done differently at runtime?

So, the algorithms executing at runtime have to be implemented in such a way that they transition from an execution mode optimized for small data to an execution mode optimized for large data in a graceful way and in an incremental way, as opposed to having a big switch. For a simple example but something that's nonetheless used heavily by systems and customers, imagine you want to sort data. If the sort input fits in memory, you'll probably use an in-memory sort, like quicksort, and the data gets loaded into the sort workspace and then gets scanned out of the sort workspace. If you have one record more than fits in memory, how much data gets written to temporary storage?

It seems it should only be one record or one page. But in the naïve implementation that might have been done under time pressure – "Let's get the release out" – there might actually be a sort that spills the entire memory content. Now, if you have a gigabyte sort space, and you spill at 100 megabytes per second, and you load it back in at 100 megabytes per second, that's 20 seconds right there. So, for one extra record, we have 20 seconds extra runtime. And customers are guaranteed to come back and say, "Bad plan chosen."

Can you give another example? That was a great example.

Well, let's take hash join. There are different versions of hash join, different versions of hybrid hash join. In particular, in terms of when you need to know how big the inputs are. And if you implement hybrid hash join from the get-go, from the start, anticipating unknown input sizes (inputs that are smaller or larger than the optimizer might have said), then the hybrid hash join should spill incrementally. So, it should start running as an in-memory join and then spill a little bit, and if necessary, spill a little bit more. So, that would be a graceful behavior.

Can hardcore database engine internals research still be done in academia?

Absolutely. Yes. And lots of people do. In fact, in every SIGMOD, every VLDB, you see a number of papers where somebody has done maybe only a twist on something previously or something fundamental. And yes, there's a lot of interesting work coming out of academia. Not everybody who can get a program to run necessarily and implicitly and immediately has interesting work. But I think there is absolutely interesting work to be done in academia but also in industrial research.

Are there any hardcore database engine internal problems where the research really needs to be done in industry rather than in academia?

I don't think so. Much of it can be done either place. When you say industry, you also have to distinguish between product groups and research groups. I think they really have different roles. But academia has yet another role. The way I see it, it's clear what a product group does. A product group produces product and either provides it as a software product on a DVD or as download, or also as a cloud service. Academics do research. They create IP and of course pass it on to the next generation. Industrial research labs have a very different role from both of them. Industrial research labs, in my opinion, should enable informed decisions.

This is something that the product groups don't do. The product groups have to execute. The product groups adopt technologies that they can adopt with a predictable effort (say in software development and testing) and with a predictable result (say in performance, efficiency, scalability, or robustness), whereas industrial research labs should take promising intellectual property and promising techniques and technologies and develop them to the point that leaders in product groups can make informed decisions about whether to adopt a technology, how to adopt a technology, or whether to skip a technology.

That's very interesting. But isn't it the product groups who have the most insight into what the pain points are for the customers, in other words, what IP needs to be developed?

Yes. But understanding the pain points, that can easily be transferred from a product group into an industrial research lab. I totally agree that the product groups should somewhat guide the industrial research labs. On the other hand, only "somewhat" because I think there is this famous quote attributed to Henry Ford: "If I had done what my customers wanted me to do, I would have produced faster horses." And I'm sure there are variants to that one.

But I think what industrial research groups also ought to do is prevent the product groups from getting scooped. So, explore outside technology that may or may not disrupt the products in some form. So, I think there too, the research labs should help the product leaders to make informed decisions, what to prototype, what to adopt, what to skip.

[…] if you have 17 join algorithms in your system, chances are you'll hardly ever pick the optimal one. In fact, you should be happy if you always pick a good one. And it's unlikely to be the case.

I like what you're saying. But isn't it true that if the industrial research lab came back and said, "You guys should really take a serious look at this disruptive technology," wouldn't the product groups not be very happy to hear that since it would disrupt their entire income stream?

Well, let's take an example. A traditional database product has a product group and a research lab. The research lab says "in-memory databases are going to be there", what the product leader might want to know is when. Also, what do we know about what the competition is already doing in terms of what public material is out there in websites or conference papers or something. And also, the next question that the product leader will ask is: what actually works? Just saying, "in-memory databases are coming, in-memory databases are coming," is not sufficient. It doesn't enable informed development decisions. It doesn't enable informed investment decisions.

At some point, the product leader has to say, "I'm not going to have five people work on a faster backup. But I'm gonna take three of them and have them work on in-memory transactions," or something like that. And that's a decision. And making that decision an informed decision, that's where the industrial research lab can create tremendous value for the industry, but also for the progress of the customers and of the applications and of whatever benefits they will provide.

Great. Young researchers would like to know what long-term hard systems problems you see. Not the hot topics but the long-term issues.

Well, that's a difficult one. And given that you are asking about long-term questions, I have a high probability of being wrong. So, I think scalability and robustness in scalability is going to be a big issue. I think we are going to reinvent a number of systems issues repeatedly. So, for example, today, we achieve robustness in scalable systems by mirroring like crazy. Every data page is written in multiple places. And if one of those places breaks down, we'll rely on the multiple copies. Now that is very expensive.

And if data keeps exploding in size in an exponential growth curve, and hardware is not growing as fast in storage capacity and processing capacity, then we might actually find that we can't have as many copies anymore. We have to do something with fewer copies. And I think that probably is going to be with us for a while.

Another problem that will be with us for a while is the problem we already talked about briefly: robust performance. I think designing efficient algorithms, making things parallel and scalable, those are trickier at times, but more manageable. Robustness is much harder, partially because it's much harder to measure. And if we don't have a clear agreed-upon metric, it's very hard to prove that my technique is better than your technique or the technique published last year.

You traditionally work on very intricate details of relational database engine internals. And this isn't an obvious match with your current employer, Google. Although of course, Google also cares about issues like fast recovery from failures. Can your results also be applied in some kind of way to Google's kind of massively parallel infrastructure?

I believe very much so. So, let's look at the query processing work that I've done and that I'm still doing. Google actually has multiple SQL engines. So, Google has multiple query optimizers and multiple query execution engines. All of them of course, designed and implemented from the get-go to be very scalable. So, efficiency, scalability, and robust performance are issues on all of these engines.

If you look at the indexing things I've worked on in the past, Google, like everybody else, will store more and more data in memory, meaning with very low latency. And in the past, a go-to on disk, a random access on disk, was considered very expensive. On a traditional disk drive, you can scan a megabyte in the time to read one byte in a random place. So, therefore, there are a number of systems that are optimized for fast scanning. And the principle optimizations for fast scannings are column stores and compression. So, in my mind, I think of column stores as optimized for disk-based data centers, disk-based data collections (traditionally, historical data collections have been disk-based). But when it comes to transaction processing or shorter history – not years of history but shorter history – and in particular looking into the future, I think more and more data will be in memory, where random accesses are much cheaper.

So, I think indexing and index-based query processing, and that means index maintenance techniques, index concurrency control, index recovery, index compression, all those things will definitely be used at Google, but also elsewhere. And I think whatever companies that are out there that Google competes on a business level, it also competes on a technology level, what internal technology is used. Google is clearly interested in in-memory processing, in-memory indexing, SQL query processing, and so on.

Think about concurrency control with many-core processors. Concurrency is an issue because there are multiple threads, multiple transactions running in any sphere of control, in any operating system instance. Concurrency control is a big issue, and I think the work I have been doing recently on precision in concurrency control – lock sizes and lock durations – can very much have an impact on Google.

Thinking about recovery and availability, obviously, Google very much depends on continuous processing of its logs. Google collects a lot of logs from online activity. And those logs need to be processed. One day's worth of log needs to be processed in less than a day, otherwise, we fall behind. Keeping that log processing pipeline and all its components up and running is very important too, You can think of all of those things as invented and designed and perhaps publicly described in the context of traditional relational databases. But many or all of those things are very much transferable into other environments.

Today's commercial query optimizers are all based on Cascades, the top-down rule-based approach to query

optimization that you put together 25 years ago. Does it surprise you that even new optimizers like Orca still use the Cascades framework?

Very much. Yes, it does. Orca actually not only uses the approach, but I think Orca is somewhat of a reimplementation of the Cascades paper[2]. And I just heard today at the SIGMOD conference here in Chicago that somebody had as a student semester project a reimplementation of Cascades. And that is now an open-source piece of software, apparently. Yes, it surprises me very much that people still follow this approach. I think this approach is very good with respect to extensibility. So, if you want to bring in a new operation into your specification algebra or into your execution algebra, then yes. Cascades is very nice because it's very extensible.

On the other hand, Cascades doesn't do anything for anybody with respect to cardinality estimation, which is really the Achilles' heel of compile-time query planning.

I don't believe that the solution for the lack of robustness in query performance will come from the planning part of query processing. I actually believe it will come from the execution part from query processing.

The other thing is I think if you look at the core of most relational queries, it is still joins. And I think the group around Pat Selinger at IBM Almaden, and their paper from 1979 is still a foundation[3]. I think Thomas Neumann has done excellent work with his advisor Guido Moerkotte and then since on extending that to more complex join predicates, for example.

If I were to build a query optimizer today from scratch, I would use dynamic programming for join optimization. And I would use a Cascades-style transformation approach for extensibility. But as I said earlier in this conversation, I would also build a query execution engine to complement my optimizer, in a way that it is very forgiving of poor plan choices.

What do you think of key-value stores?

I think key-value stores have their place in the scheme of things. Key-value stores come in a wide variety of scalability, capability, and so on. At some places, they are the right tool. And that's what it's all about, choosing the right tool. At some other places, they are not. Personally, I am very convinced that application programmers want serializable transactions, meaning application programmers have the freedom, the liberty, the simplicity of thinking whatever transaction they run is the only thing going. I think that's a powerful paradigm. Some people strongly agree with me. Some people strongly disagree with me. And that's okay. I happen to have one belief. There you have it.

Some key-value stores are better about it than others. And I think some people trade performance for concurrency, for cleanliness of transactions. As I said, I usually would forego performance and scalability if I can get cleanliness of the application model. But then I think we, as data engine experts, should try to make the clean application programming model highly efficient in the engine.

So, I mentioned earlier concurrency control, the granularity of concurrency control, the duration of locks, how many false conflicts do we detect and treat them as if they are conflicts. In my concurrency control work that is basically the theme: avoiding false conflicts. And I think there is probably a factor 100 in that.

Wow. Okay. You teach a one-week course on – we won't pigeonhole it. We'll just say database engines – every year at Dagstuhl. In this day and age of education over the internet, why don't you just record your class and leave it on the web for posterity?

Well, there are many reasons for that. I think the students, typically fresh masters graduates, get much more out of it if it's interactive. Even when I was teaching undergraduates, I always was trying to learn names, basically have conversations rather than lectures. So, I think it's much better for the students if it's interactive. I think it's also much better for the students if they in some sense experience, what for many, is the first international event.

And *Dagstuhl* is a very nice and protective environment. For many of the participants, that's a very positive experience. Personally, I enjoy it very much. Yes, I miss teaching. I used to like it very much.

[2] Goetz Graefe: The Cascades Framework for Query Optimization. IEEE Data Eng. Bull. 18(3): 19-29 (1995).

[3] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34.

And so, this is my outlet. And I love *Dagstuhl*. In fact, I'm on one of their boards, so I have to go there for their board meetings at least once a year.

Do you have any other words of advice for fledgling or midcareer database researchers?

Well, what advice? Never give up. That's really the advice I have because there have been a number of times where things have not gone well in my career. I had to leave a university because clearly my tenure was going down the drain. In retrospect, they probably would be happy to have had me. I think other things have not gone my way. You just keep plugging away, and you show them. And that would be my advice.

Work on real problems. Solve problems that you know exist. And then have confidence that you can solve them and keep working on them.

Work on problems nobody cares about because in particular, if you don't have a large group, that's the best way to make progress without fierce competition. For example, at Hewlett-Packard, I felt at times I was the only database expert in Hewlett-Packard Labs. And so, I worked on stuff like concurrency control and join algorithms. And I knew there wouldn't be competition. If I get it published this year, get it published next year, nobody cares. Nobody will scoop me because nobody was working on concurrency control and join algorithms. So, just keep plugging away, and you'll get there.

If you magically had enough extra time to do one additional thing at work that you're not doing now, what would it be?

I would really, really love to have a team to implement a new system that actually is innovative by simplicity. Simple is absolutely important because if it's not simple, I don't understand it. And every system I know has gotten so unbelievably complex. And people revel in the complexity, it feels to me. Building something really simple, that would be fun. But it would require a small team to build something that is still robust say, against data loss, but also robust in terms of query performance.

If you could change one thing about yourself as a computer science researcher, what would it be?

Perhaps I would have stayed an extra year in graduate school and had learned about artificial intelligence and basically had done more with it. I mean, I went through graduate school in four years, which was fast. And I think maybe if I had stayed an extra year, that could have been fun.

Thank you very much for talking with us today.

It's been my pleasure.