Ryan Johnson: recipient of the 2012 ACM SIGMOD Jim Gray Dissertation Award

by Marianne Winslett and Vanessa Braganholo

Ryan Johnson
http://www.cs.toronto.edu/~ryanjohn/

Welcome to this installment of ACM SIGMOD Record's series of interviews with distinguished members of the database community. I'm Marianne Winslett, and today we are in Phoenix, site of the 2012 SIGMOD and PODS conference. I have here with me Ryan Johnson, who is the recipient of the 2012 SIGMOD Jim Gray Dissertation Award for his thesis entitled "Scalable Storage Managers for the Multi-Core Era". Ryan's advisor was Anastasia Ailamaki. His PhD is from Carnegie Mellon, and he is now a professor at the University of Toronto. So, Ryan, welcome!

What was your dissertation about?

Well, the short version is making databases run really fast on modern hardware. Especially changing modern hardware, because we are in a time where designs are under huge flux and the things that were problems let's say every ten years, are now problems basically at every hardware generation. So there is a lot of work for the database engine to keep pace and to insulate the rest of the database community from all of these disruptive chambers of the lower levels.

So what kinds of things do you have to change in the database core to keep up with the greater *parallel processing that we are now seeing?*

Honestly, you have to be willing to change about anything, though you try to change as little as possible. The usual culprits are things that were centrally managed in the past. Things like locking and logging are huge culprits because they are central points of design. And so finding ways to either distribute those, or make them at least act like they are distributed without giving up the semantics, is the key to getting the scalability we need.

What does it mean for locking to be distributed?

So, by design, the database engine wants a central point where we can find the state of the world. And locking is designed to give you that, for the semantics of the applications that rely on it. So the application can say "I locked this value, nobody else is going to change this value". What, exactly, that means is mostly application dependent. The database's job is just to enforce the lock itself. The semantic of this globally visible, you know, one state of the world for each lock, means that you kind of need the central point, but it turns out that usually people aren't changing the protected value. So the trick was to identify when the bottleneck comes out of read-only values, and then play dirty tricks under the hood. Semantically, at the visible level, the system is still exactly the same as before, but under the hood, the lock protocol has been spread out so that these hot spots no longer arise. In practice, that works really well because a database application where people are fighting over updates doesn't scale for other reasons anyway.

Be willing to jump on a good thing if it comes.

Do you use a technique where you sometimes abort transactions later on because they had a conflict that you detected later on?

So that is standard practice for database engines, I didn't really change that part. What I was looking at was when a database says "I want to protect this from updates by other transactions: I'm only going to read, I just don't want the rug pulled out from under me while I am doing it". And everybody does this, it turns out. They say, "lock the database so that nobody deletes the database", "lock this table so that no one deletes this table", and it's those locks that were the bottleneck. Because everybody is doing it! And the time it takes to find out "is the lock compatible?" was long enough to create a bottleneck. So the trick was to say, since everybody does it, we'll assume that whoever is next will do it, we'll just keep the lock. It's read-only, it's not like anything has changed. The agent thread that was executing a transaction will just hold onto the lock, and hope that the next transaction can use it. And 90+% of the time, we're right, and now you've completely cut the [central] lock manager out of the picture, and so it is no longer involved.

Was there something for logging that you had to do?

Logging was a different one. Locking was theoretically parallel but in practice it wasn't because of the low level implementation. Logging is the opposite. It's supposed to be serial, right? We want the one global history that Mohan gave us [with ARIES], so we can say the state of the world from all viewers is the same. But under the hood, we really want that to be parallel so that more than one thing can happen at a time. The trick there was to exploit a technique called linearizability, which essentially says we can play tricks with time. And so everyone will agree in the end on what the order was, but it's not necessarily the order that things actually happened in. But since we all agree, that's okay. Doing that relaxed the constraints on the implementation enough that we could do some things under the hood to get the parallelism that we needed. So at this point, you're going to run into other problems before the log becomes a problem again.

Has your dissertation work had impact?

It has. So first of all, the storage engine [Shore-MT] is in use by several researchers at various institutions, academic and industrial, and so that has been really nice. We debated whether to open source it or not, and in retrospect, it was a really good idea. People are using it and it is helping their research. Also, developers for the various open source databases have mentioned using various things [from my dissertation work]. I talked with people from MySQL, people from Berkeley DB, recently the PostgreSQL people talked about picking up the logging stuff, and so it has been really cool to see people actually interested in the work and trying to move forward with it.

You have to know why things are happening, and not only why they're happening, but why they didn't work, and why they did.

What about companies?

The commercial database vendors don't talk an awful lot about what's under the hood. I am in active talks with both Oracle and IBM, so they are interested in the kinds of things I do, but whether they are using precisely what I've already done, I don't know.

It's amazing to me to see such a core traditional database thesis being honored in this way because I discovered that the young people in our field, most of them, they don't know how a database works. Indexing, maybe. Read and write locks, yes. But anything beyond that, they have no idea. So do you think this is a problem, or is it okay because the field has expanded so much, people can't all know where we started?

This is a tricky one. It becomes a challenge sometimes because you submit all of these systems in papers, and don't necessarily get an audience who has any idea or interest in what it was [you did]. In the end, I decided this means that we're being successful. A small handful of people can make these problems go away, and nobody else has to worry about them. But it doesn't always work, right? We have to have some guidance from the people up higher of what they need so we can provide it. They need guidance from the people down lower about what's just not going to work, to shift their expectations a little bit. So the database engine is really this two sided thing:

there's the level that faces the user above, and there's the level that faces the hardware below. I'm on the hardware facing side, but we still have to talk to each other.

Is there any advice that you'd like to share with our readers or viewers who are working on their PhDs right now?

One final thing that I really became more aware of as got toward the end of my PhD was that you have to be willing to cross boundaries into nearby disciplines. It is not enough to be an expert in one narrow little tiny slice because it is all mined out, to be honest. Database engines are not new things, we've had them for 50, 60 years, and so if you want to really be able to solve these problems, you've got to be willing to bridge gaps between nearby things, and very often it turns out the problem actually isn't in the database engine. Some of the work that I enjoyed the most was actually going out into the operating system, or dealing with other nearby things, or talking to the SPAA community about distributed algorithms. Those things are what really enrich the research when you are able to go across boundaries and not be scared of new material.

Is there something you now know that you wish you had known earlier in your research career or perhaps during your job hunt that you can share with our younger readers and viewers?

So, I'll rephrase that a little, because I had a great advisor, who taught me early on things that I am really glad that she taught me. One of them was that you always want to understand what's going on. You get this result, and there is some hiccup or some bump in the data that you don't understand, you don't ever let that slide. You have to know why things are happening, and not only why they're happening, but why they didn't work, and why they did. Very often we get people with results were they say "we turned this knob and performance went up, let's declare victory and go home". But we don't know why it worked. We don't know how long it is going to work, or whether we just got lucky. So if you can point to things besides just performance and say, "look, I have analyzed the system and here are reasons *x*, *y* and *z* why we know this problem is gone and it's not going to be back". That gives you a much tighter control over your results because now you know why they are doing what they are doing. And that leads to whole new areas. One of my papers came from "why in the world would that happen that way?". It would have been a huge mistake to let that [question] slide.

Other things for the job hunt, I had a kind of odd job hunt because I only applied to one school. It was a surprise thing, and I said, "I'll try it, and if it doesn't work, I'll go on the job market next year". And it turned out I loved the place. So one thing I would say is "be willing to jump on a good thing if it comes". Don't keep searching after you've found something that is good. Sometimes we get stuck on this "I have to have the absolute best" and we let so many good things go by that we look back and regret it. So I am really glad that I have been able to jump on things early and then look back and say, "yeah that was actually a good idea".

Excellent advice. Thanks so much for talking with me today.

You're very welcome.