Stratos Idreos Speaks Out on Database Craking

Marianne Winslett and Vanessa Braganholo

Stratos Idreos
http://stratos.seas.harvard.edu/

Welcome to 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 Stratos Idreos, who is the 2011 recipient of the SIGMOD Jim Gray Dissertation Award for his thesis entitled Database Cracking: Towards Auto-tuning Database Kernels. Stratos's advisors were Stefan Manegold and Martin Kersten, and his PhD is from the University of Amsterdam. Stratos is currently a tenure-track researcher at the Dutch National Research Institute for Mathematics and Computer Science (CWI)[1]. So, Stratos, welcome!

[1] Stratos is currently an Assistant Professor at the Harvard University.

Tell me what your thesis is about.

My thesis introduced the concept of database cracking. The main idea is that every query that comes through the database system will be used as an advice on how data should be stored on disk and on memory. So basically, the system creates indexes incrementally and on the fly during query processing. Normally, database systems would need enough idle time and workload knowledge to create indexes. Now, with database cracking, indexes are created automatically without you having to worry about all these preparations.

So when you say incrementally, do you mean you create the whole index while queries are running, or you create part of a traditional index?

Exactly, that is a very good question. So, creating the whole index while queries are running, this is online indexing. There's a couple of works by Surajit Chaudhuri and Nico Bruno at Microsoft that do online indexing. In addition, there is the work by Alkis Plyzotis which came at about the same time. Our work is about incremental indexing. We create only parts of the index during query processing. So, let me give you a more representative example. I create part of indexes within select operators, for example. So if you have a select operator of a query that says give me everything from this table where values of attribute A are between 20 and 30, then we would take the column of attribute A and we would split it in three parts: from 0-20, from 20-30, from 30-whatever. Then you have introduced range partitioning by splitting the table in three pieces, and that is enough information to improve future queries.

So, then if you want to use an index in the future, you start by checking if the data is covered…

If this partition exists, you can use it, you can explore it, you can refine it even more. So these little pieces that you create, they become smaller and smaller with every other query. And every other query introduces more and more partitioning, which means more knowledge about how data is laid on disk, and then you can explore it. And by pieces becoming smaller, performance becomes better.

So when you say smaller, what do you mean smaller? I would think they would become larger over time.

Larger in terms of how many they are, smaller in terms of how many tuples they have inside.

I see, so they get divided into finer grained courses.

Yes, and if you think about it, at every range select operator you have to touch at most two pieces, because you only have to check the boundaries of the range, and that's at most two pieces. And by pieces becoming smaller, you have to analyze less tuples with every other query.

> ***I stopped taking rejected [papers] reviews very religiously.***

Doesn't this make query optimization harder?

Yes and no. No because you have chosen to always use indexes. So every query will use indexes, there is no decision about that. You blindly go and use the same database plan with every query. You could think about it as if you always created clustered indexes, basically. It's not secondary indexes.

I see. When you say it is not secondary indexes, do you mean you rearrange the data on disc to match…?

Yes. We rearrange the actual data. We create copies of the data, and we rearrange these copies. So at the first time that you query, for example, if you want to select over attribute A, we create a copy of this column and we start rearranging this column. And then every query that wants to select on A will go directly there, and won't touch the base data anymore.

So, later queries on attribute B make another copy with B, and not the whole tuple, just the tuple ID is there?

Exactly. It's attribute B and the tuple ID.

But what about updates?

Updates, that's a tricky business. But what we do with updates is that we defer them. When updates come, we just keep them aside. And we only merge them when a relevant query comes. So let's go back to the previous example: attribute A, between 0-10, 20-30 and so on. Then if another query comes and says, okay, I want values between 20-25, if and only if there are pending updates within this range, then we merge them on the fly during query processing. So the select operator would not only fine grain the partitioning information, but it would also merge updates.

And if I understood correctly, if the query is over attributes A and B, you'll have a little index that's just for that range of A and B?

Yes.

Hmmm, very interesting. So how did you show that's better than the alternatives?

First of all, let me clarify that this kind of ideas, this kind of research, is applicable for exploratory dynamic workloads. So in the case that you know exactly what you are looking for, you have enough idle time to prepare your indexes for that, you should not be using database cracking, there's no sense. But in the case where you don't have enough knowledge about the workload, and you don't have enough idle time to prepare, then is when you should be using database cracking. So what we always do in our experiments is we compare database cracking with a plain, non-indexing approach, where you have to scan your data, and we always compare it with the perfect indexes, which in the case of column-stores is when you have basically sorted arrays. And that is the equivalent of offline indexing, in this case, because, in order to sort an array, you need time to do the sorting, and you need to know that this array is useful when sorted. What we typically see in these examples is that the performance of database cracking starts with the first query being almost as expensive as a scan (just a little bit more expensive), and then it quickly improves performance, and after a few queries, it reaches the optimum performance of an index. But the offline indexing approach takes typically 10 times more in order to create the index. So if you don't have idle time, you first have to pay this 10 times more overhead.

Ok, that's interesting.

> ***Take good care of yourself, (…) do some physical activities.***

Maybe a more representative example would be queries of TPC-H, for example. In order to create the optimal indexes for the columns in TPC-H in this particular machine that we used and everything, we needed about 3 hours. With cracking we could answer all queries, getting to optimum performance in a matter of seconds, basically.

And then, so if you just did cracking with no previous knowledge of the TPC-H workload, versus if you had created the perfect indexes beforehand, what's the difference in performance at run time of the transactions?

So, we havent't studied extensively the performance of transactions, we typically do only analytical read queries. But the difference compared to the optimal index is basically zero. You reach the optimal performance. You don't expect optimal performance as of query one. In the case of TPC-H (it is actually a good case for us because the workload is skewed), you reach optimal performance in a matter of 5-10 queries, and the good point is that as of query number two, you are way below the performance of a no index approach. But then as of query 5-10, you reach the optimal performance of a perfectly tuned database. Now, if you devise micro-benchmarks, where you have random workloads basically, this optimum performance comes after thousands of queries, not after 5 or 6.

Do you have any words of advice for today's PhD students?

I would have many. My main lesson that I try to remember now after my PhD is that I stopped taking rejected [papers] reviews very religiously. So one big mistake that I think that I made over the years is that sometimes I got reject reviews (and I got many of them), and then I thought that "okay, I should react very very seriously based on this review", and sometimes I ended up basically just destroying papers and making them very dense, just because I was trying to put every little detail in there. So I think this would be good, although we should take rejects very seriously, and put comments to use, but maybe we should also take a step back, and think about it again.

Is there another piece of advice you would like to share?

Yeah, I'll say that it's not only about research, we should also take good care of yourself as well, so maybe sometimes take a step back, don't do so much research, do some physical activities.

Very good! Thank you very much for talking with us today!

Thank you!