PROFESSOR: What we're going to talk about today is in the previous class, we did a lot of examples on some data sets. And they were artificially generated data sets, which is a requirement in terms of if you want to do anything with big data and really show stuff, it can be really difficult to pass around terabytes and terabytes of data. So having data generators that generate the statistics that you want or approximate the statistics that you want are good to have. We had a previous lecture on power law graphs and talked about a perfect [AUDIO OUT] is very useful, and I encourage you to use that. It's a useful analytic tool.

This one, we're going to talk about Kronecker graphs and generation. Kronecker graphs are used in the largest benchmark in the world for benchmarking graph systems. And then finally, I'm going to talk a little bit about performance. What are the things you should be aware of when you're doing systems and building systems from a performance perspective, the kinds of things that are essentially fundamental limits of the technology.

So moving forward, let's talk about the Graph500 benchmark. So the Graph500 benchmark was created to provide a mechanism for timing the world's most powerful computers and see how good they were on graph type operations. And so I was involved and actually wrote some of the initial code for this benchmark about 5 or 10 years ago. And it's now become a community effort, and the code has changed, and other types of things.

And it's mainly meant for dealing with parallel in-memory computations, but the benchmark is actually very useful for timing databases or any types of things. You can do the exact same operations. So it generates data, and then it has you construct it in a graph, and then it goes on to do some other operations. And so we've actually found it to be a very useful tool as a very fast, high-performance data generator. And if you want to time inserts of power law data, it's very useful for that.

This is some older performance numbers. But this just shows essentially here on a single core of computing capability, the number of entries that we're inserting into a table and the inserts per second that we're getting. So the single core performance here for this data into

Accumulo, it's about 20,000 inserts per second. So there's a one core database, very, very small. As you've seen-- we've even showed last week on modern servers where we're getting several thousand inserts per second on standard databases.

I'm comparing this also with just D4M without a database. If I just have triples and I construct an associative array, how fast does it do that? And so obviously, D4M is limited by how much memory you have on your system, but it just show you the constructing and associative array. And memory is faster than inserting data into a database by a good amount. And so again, if you could work with associative arrays and memory, that's going to be better for you. Obviously, though, the database allows you to go to very large sizes. And one of the great things about Accumulo is this line stays pretty flat. You can go on, out, and out, and out, and out. That's what it really prides itself on, is that-- as you add data, it might degrade slightly, but if you're inserting it right, that line stays pretty flat.

In the last class, we did show how to do parallel inserts in parallel D4M. And this shows parallel inserts here on a single node database, D4M itself basically taking this number and scaling it up and also into the database. And so here a single node, single core database who are getting 100,000 inserts a second with that. And so that just shows you some of the parallel expectations, and when you should use parallel versions databases. Again, if you can be in memory, you want to be in memory. It's faster. If you can be in parallel memory, you probably want to be in parallel memory. That's faster. But if you have really large problems, then obviously, you need to go to the database. And that's what that's for.

The data in the graph 500 benchmark is what we call a Kronecker graph. It basically takes a little, tiny matrix, a little seed matrix-- in this case, g-- and you can imagine what it looks like here. Imagine this is a two-by-two matrix here. You can sort of see, maybe, a little bit of the structure. And then it does a Kronecker product of that on out to create and adjacency matrix. By creating this adjacency matrix, that creates the graph.

And as a result, it naturally produces-- this is the result. It produces something that naturally looks like a power law graph here. And this just shows you the dots, shows you the power law degree distribution of that. You can see the slope there generated from this. I should say one of the powerful things about this particular generator is you don't have to form the adjacency matrix to construct the graph. So it doesn't require you to build this gigantic matrix. It generates the edges atomically.

If you have many, many processes all doing this, as long as they set their random number seeds to different locations, they'll all be generating perfectly consistent edges in a larger graph. So people have run this on computers with millions of cores to do that. Because the graph generator is completely-- parallel allows you to essentially create a giant graph.

And also, the Kronecker graph does generate this structure here, which is these bumps and wiggles. And that actually corresponds to the power k. So you can count the number of bumps. We have 1, 2, 3, 4, 5, 6, 7, 8. So this would be a g to the 8th graph. So that structure is visible there, which is an artificial feature of the data.

And so one of the things that's actually occurred now-- because people have been actually trying to use this generator because it's so fast and efficient to simulate graphs. And we've basically said to people, not really designed to simulate large graphs, because it does create these artificial structures in there. And that's the reason we develop the perfect power law method. Because that's a much better method if you're actually trying to simulate graphs for doing that. Again, this method has these advantages in terms of being able to generate them.

Another advantage of these Kronecker graphs-- and so in this case, B is my little matrix here that I'm going to do Kronecker products of. In fact, let me remind you about what a Kronecker product is. So if I have here a matrix B, that's nb by nb, although these don't have to be square, and another matrix C that's nc by mc doesn't also have to be square. And when you do the Kronecker product, basically what you're doing is essentially taking c and multiplying by the first element, and having essentially a block here.

And so it's a way of essentially expanding the matrix in both dimensions. And Jure Leskovec, who's now a professor at Stanford, essentially developed this method back in 2005 for creating these matrices which allow you to very naturally create power law matrices, and even had tools for fitting them to data, which is a useful thing. Again, its biggest impact has been on generating very large graphs, power law graphs very quickly.

The Kronecker graph itself has several different types that I call here, basically, explicit, stochastic, and an instance. So if I just set the coefficients of the Kronecker graph to be ones and zeroes and I Kronecker out, I will get structures that look like this. So this is a Kronecker graph that essentially is a bipartite, essentially starting with a-- what you can call a star graph and then a diagonal. So that's 1. That's g1. Then I Kronecker product again, g2 and g3, and I get these different structure 0 1 matrices. And so it's only 0's and 1's, and so the structure and

the connections are very obvious.

And this is actually useful for doing theory. So you can actually do a fair amount of theory on the structure of graphs using the 0 1 matrices. And in fact, there's a famous paper by Van Loan on Kronecker products. And basically, the paper is full of identities of Kronecker products that-- the eigenvalues of the Kronecker product of two matrices is the same as the Kronecker product of their eigenvalues, and a whole series of those things. And so what's nice is if you compute something on just the small matrix g, you can very quickly compute all the properties of the Kronecker product matrix, and that makes it very nice for doing theory. And so there's times when you want to understand the structure of these things. Very useful theoretical tool for doing that.

If instead, we use our C graph, instead of it being 1's and 0's, it contains, say, numbers between 0 and 1 that are probabilities, the probability of creating an edge between those two vertices. And we multiply that on out. You get, essentially, a giant probability matrix showing the probability of edges. These are the same, but here we see we have differences. We don't just have 0's and 1's. We have values between 0's and 1's.

And then using the Kronecker generation technique, we essentially draw instances from this probability matrix over here. So this allows us to randomly sample. So when you're talking about Kronecker graphs, there's three different types here. This is very easy to do theory on. You can do theory on this, too, but it tends to be a little bit more work-y. And then likewise, when you're doing simulations, usually you end up with Kronecker graphs like this.

As an example of the kind of theory we can do, one of the nicest graphs to work with are bipartite graphs. So a bipartite graph is two sets of vertices, and each vertex has a connection to every vertex in the other set, but they don't have any connections amongst themselves. And so here, I'm showing you a bipartite or star graph, so basically one set of vertices here and another set of vertices here. And all the vertices have connections to those vertex, but they have no connections themselves. And this is the adjacency matrix of that graph.

And here is another bipartite graph. It's a 3 1 set. And if I Kronecker product two bipartite graphs, the result is two more bipartite graphs. So as we see here, we've created this one here and this one here. And so in my notation, I'll say B 5, 1 is a bipartite. That's basically-- I'm saying I have a matrix that's a 5, 1 bipartite graph, 3, 1 bipartite graph. And within some permutation, they produce a matrix that is the union of a 15, a 1, and a 3, 5.

And this result actually was first shown by a professor by the name of Weichsel in 1962. And there was actually a little flurry of activity back in the 1960s with this whole algebraic view of graph theory, which was very productive, but then it all died off. And after the lecture, I can tell you why that happened. But now it's all coming back, which is nice. So it took us 50 years to rediscover this work. But there's actually a whole book on it with lots of interesting results.

But then this just shows you the general result here. For any two bipartite graphs, if I Kronecker product them together, they are under some permutation equal to two other bipartite graphs. And this naturally shows you how the power law structure gets built up when you Kronecker product these graphs. So basically, you had this and this. And there, we basically-- that's the super node vertex here, this 1, 1 vertex. And these are the lower degree pieces. And then these other vertices here are the singleton vertices. So you can see as you naturally product these bipartite graphs together, you naturally create these power law graphs.

An example-- I won't really belabor the details here-- you can compute the degree distribution. And this is all-- there's a long chapter on this in the book. Basically, you can compute analytically the degree distributions of Kronecker producing bipartite graphs. This shows you the formula for doing that. And here is the actual degree distribution with the binomial coefficients for that. And so a very useful thing. You can a priori do that.

This shows you the results. So if I have a Kronecker product of a bipartite graph n equals 5 comma 1, and I take it out to the 10th power, this is the result is this blue line. And what you see is that you get-- if you connect the last vertex and this vertex-- essentially our poor man's alpha-- or connected any one of these other sets here, they have a constant slope of negative 1. So that's a theoretical result is that they have this constant slope. And that's true down here, k to the 5, other types of things. So we do get this bowing effect. So it's not perfect with respect to that. But you can see how this power law is baked in to the Kronecker generator, which is a very nice thing.

This shows a sample. We basically created a 4 by 1 bipartite graph and then sampled it. And so this shows the stochastic instance. There's about a million vertices here. Shows you that. And then these curves here show-- when we take the stochastic graph and analytically compute what the expected value of every single degree is assuming a Poisson distribution, and you can see that we get pretty much a perfect fit here all the way out, except to the very end here.

And this is where you get some pretty interesting statistics going on. In theory, no one vertex-- this is your distribution. No one vertex should actually be attracting this many nodes. But you have so many vertices with one of them that have this problem. When you sum them all together, one of them gets to be chosen to be the winner. And it's what pops it up here. So all the way out to the-- our fit is very good all the way out to the super node, and we have this difference here. And this is where the Poisson statistics begin to fail. So again, lots of opportunities for a very interesting theory there.

So that's just pure bipartite graphs. But bipartite graphs have no inter-group connections. So we can actually create something that looks like a power law graph and see all the community substructure very clearly. But those communities are not connected in any way. And so to create connections to the communities, we have to add the identity matrix down the diagonal. And that well then now connect all our sub bipartite graphs together.

So this just shows here-- so we basically take our bipartite graph plus the identity, and that's essentially this combinatorial sum here of the individual bipartite graphs there. Again, where in quotes, it means that there's a permutation you need to actually make this all work out.

We actually do the computation. This shows the 4, 1 bipartite plus an identity bipartite graph out to the fourth power. So you see here you're going to get 1, 2. You can compute it on out there. And what you see is this recursive structure, almost fractal like structure. So that's a nice way to view it. That's one way to see it. But you don't really see the bipartite substructure in there. It's hard to see what's going on there.

Well, since this is analytic, we can actually compute a permutation that says please recover all those bipartite substructures. And so in the software, actually, [INAUDIBLE], we have a way of basically computing a permutation of this that basically regroups all our bipartite groups here. And then you can see all the bipartite cores or bipartite pieces, and then all the interconnections between those core bipartite pieces.

To give you a better sense of what that really means, here, basically we have b to the i to the third power here. That's what it looks like. If we just subtract out the first and second order components, then we're left with these are the interconnecting pieces. We can see that much better when we permute it. So here is the permutation of just the bipartite piece, and that shows you these core bipartite chunks that are in the graph. We then do the second term, which is b kron b kron i.

So this is the second. Now you can see that this creates connections between these groups. And then likewise, if you do the next one-- so b kron i b, that shows you the connections between these groups, and then ibb shows you the different connections in these groups. So each set of-- when we take b plus i and multiply it out and look at all those different combinations of the different multiplies there, there's the core piece, which is just b kron to the 3, and that creates this core structure here, which is these dense pieces, and then all the other polynomials are essentially creating interconnections between that. So you can get a really deep understanding of the core structure and how those things connect together, which can be very interesting.

But we can compute these interconnected structures fairly nicely. Here's a higher order example going out to the fifth. But we can compute the structure, this chi structure here, which is essentially a summary of these, by just looking at a two-by-two. So we can compute all the structures of the interconnections by just looking. Since we have a bipartite thing, we can collapse all those pieces down and just have a two-by-two. So we have here a little tiny version of this structure here. It shows you that. Likewise, this structure here is summarized by that, and this structure here is summarized by that.

So you can get complete knowledge, not just at the low level scale, but at all scales if you wanted to say I want to look at the blocks and how the edges are connected in detail, or if I just want to consider things in big blocks and connect them, I can do that very nicely as well. And again, we can compute the degree distributions as well. So this just shows b kron to the k, and then b plus the second order terms, and then moving on out. And actually, what you see is by adding this identity matrix, all it does is slide the degree structure over. It also does change the slope a little bit here by adding this identity matrix along. So it's steeper than the original.

And you can do other types of things. Here's something we did in iso-parametric ratios. I won't belabor that. But it essentially is a way of computing the surface to volume of subgraphs. And we can compute this analytically. And we see that the iso-parametric ratio of a bipartite subgraph is constant, but the half bipartite graph has this property here.

And just to summarize here-- and this is done a great deal in the chapter-- this just shows different examples of the theoretical results you can compute for a bipartite graph or a bipartite plus an identity, the degree distribution. Betweenness centrality is a fairly complicated metric. We actually haven't figured it out for this one, or I didn't bother to figure it out for this

one. I can compute the diameter very nicely. You can compute the eigenvalues. And the iso-parametric are just all examples of a kind of theoretical work you can do. And again, I think it's very useful if you want to understand the substructure and how the different parts of a power law graph might interact with each other.

So that just talks about Kronecker graphs and what are the bases of these benchmarks. And now I'm going to get into some benchmarks again themselves. So this is just to remind folks. This just shows some examples of when you're doing benchmarking of inserts into a database. Normally, when you do parallel computing, you have one parameter, which is how many parallel processes are you running at the same time. And so most of your scaling curves will be with respect to, in this case, the number of concurrent processes that you're running at one time here. I think we're going up to 32 here. That's the standard parameter in parallel computing.

When you have parallel databases involved, now you have a couple of additional parameters, which just make it that much more stuff you have to keep in your head. Essentially, in this case, Accumulo calls its individual data servers tablet servers, so you have to be aware of how many tablet servers you have. So that's another parameter. So you have to create separate curves here.

This is our scaling curve with a number of ingest processes into one tablet server versus number of ingest processes into six tablets servers. And as you can see, not a huge difference here at the small end. But eventually, the one tablet server database levels off while the other one keeps on scaling. And so this is very tricky. If you want to do these kind of results, you have to be aware of both of these parameters in order to really understand what you're doing.

There's also a third parameter which is generally true of most databases, and Accumulo is no exception, which is that you're inserting a table into a parallel database, that table is going to be split amongst the different parallel servers in the database. And that will have a big impact on performance. How many splits you have-- because you're not going to be able to take advantage of-- if you have fewer splits than the number of database server nodes, then you're not taking advantage of those. And then how balanced those splits are. Is the data going in uniformly into those things?

And so this just shows you an example of an 8 tablet server Accumulo doing the inserts with no splits versus doing it, in this case, with 35 splits. And you see there's a rather large impact

on the performance there. So D4M actually has tools. We didn't really go into them. But if you look at the documentation, we're allowing you to do splits. The databases we've given you access to are all single node databases, so we've made your life, in a certain sense, a little bit easier that you don't have to worry about splits. But it is something-- as you work on larger systems, you have to be very aware of the splits.

Another thing that's often very important when you're inserting into a database is the size of the chunks of data you're handing to the database. This is sometimes called the block size. This is a very important parameter. Blocking of programs is very important. Probably the key thing we do in optimizing any program is coming up, finding what the right block size is for a system. Because it almost always has a peak around some number here.

And so this just shows the impact of block size on the performance rate here for Accumulo. And the nice about D4M, there's actually a parameter you can set for the table which will say how many bytes I want to set. I think the default is 500 kilobytes, which is a lot smaller than I would've thought, but it's a number that we found tends to be fairly optimal across a wide range of things. Typically, when you do inserts into a database, you're like, well, I want to give you the biggest chunk that I can and move on. And Accumulo and probably a lot of databases actually like to get data in smaller chunks than you might think. Basically, what's happening then is if you're giving it the right size chunks, then it all fits in cache, and any sorting and other types of operations it has to do on that data can be done much more efficiently.

So this just shows another parameter you have to be worried about. If you do everything right-- and I showed these results before-- these are the kind of results you can get. This shows essentially on an 8 node system-- fairly powerful, 24 cores per node getting the 4 million inserts or entries per second that we saw, which is, to our knowledge, again, the world record holder for this.

That talks about inserts. Another thing that we also want to look at is queries. So as I said before, Accumulo is a row-based store, which means if you give it a row key, it will return the result in constant time. And so that's true. So we've done a bunch of queries here, lots of different concurrent processes all querying at the same time. And you can see the response time is generally, they say, around 50 milliseconds, which is great. And this is the full round trip time. A lot of this could have been network latency, for all we know. So that's what you expect. That's what Accumulo does. If you do row queries, then you get constant time.

Now, as we've talked about before, we do our special exploded transpose schema, which essentially give this performance for both row and column queries. But if you have a table where you haven't done that and you are going to query a column, it's a complete scan of the table. And so you see here-- when you want to do a column query, the performance is just going to go up and up and up, because they're all just doing more and more scans, and the system can only scan at a certain rate. So that's, again, the main reason why we do these special schemas is to avoid this performance penalty.

So finally, in talking about performance, let's talk a little bit about D4M performance itself. If you write your D4M programs optimally-- and this sometimes takes a while to get to, because usually you don't quite have the-- or I should say, in the most elegant way possible, again, this sometimes requires some understanding. Most applications end up becoming a combination of matrix multiplies on these associative arrays.

Now, it's usually not the first program I write, but usually, it's the last one I write. It finally dawns on me how to do the entire complicated analytic and all its queries as a series of special sparse matrix multiplies, which then allows us to maximally leverage the underlying libraries which have been very optimized to do-- it's basically not calling any of my code. It's just calling the MATLAB sparse matrix multiply routine, which is further calling the sparse BLAS, which are heavily optimized by the computing vendors, and likewise are calling sort routines.

So the thing you have to understand, though, is that sparse matrix multiply has fundamental hardware limits depending on the kinds of multiplies you're doing here. And we have a whole program that basically times all these and generates them. So if you ever want to get what your fundamental limits are, we have a set of programs that will run all these benchmarks for you and show you what you can expect.

And this shows you, basically, the fraction of the theoretical peak performance of the processor as a function of the total memory on that processor. So this is all single core, single processor benchmarks. So as you can see here, this is dense matrix multiply. And it's 95% efficient.

If I was to build a piece of special purpose hardware and a memory subsystem for doing dense matrix multiply, it would be identical to the computers that you all have. Your computers have been absolutely designed to do dense matrix multiply. And for good or bad, that's just the way it is. The caching structure, the matrix multiply units, the vectorization units, they're all

designed to-- they're identical to a custom built matrix multiply unit. And so we get enormously high performance here, 95% peak on dense matrix multiply.

Unfortunately, that hardware architecture and everything we do it for dense matrix multiply is the opposite of what we would do if we built a computer for doing sparse matrix multiply. And that's why we have, when we go from dense matrix multiply to sparse matrix multiply, this 1,000x drop in performance. And this is not getting better with time. This is getting worse with time.

And there's nothing you can do about it. That is just what the hardware does. Now, it's still better to do sparse matrix multiply than to convert it to dense, because you're still a net win by not computing all those zeros that you would have in a very sparse matrix. So it's still a significant win in execution time to do your calculation on sparse matrices if they are sparse, but not otherwise.

And then-- so that's that. This shows the associative array. So the top one is MATLAB dense, MATLAB sparse, D4M associative array. And we worked very hard to make the D4M associative array be a constant factor below the core sparse. So you're really getting performance that's pretty darn close to that. And so generally, doing sparse matrix multiplies on the associative arrays works out pretty well. But again, there is still about 21% efficient of the hardware.

And then the final thing is we have these special sparse matrix multiplies where we concatenate the keys together. If you remember way back when we did the bioinformatics example, we could do special matrix multiplies that essentially preserved where the data came from. If we correlated two DNA sequences, we would have a matrix that would show the IDs of the DNA sequence. And if we did these special matrix multiplies, the values would then hold the exact DNA sequences themselves that were the matches.

Well, there's a performance hit that comes with holding that additional information, all of this string information. It may be one day we'll be able to optimize this further, although we've actually optimized it a fair bit already. And so there's another factor of 100. And it looks like this is just, again, fundamentally limited by the speed at which the hardware will do sparse matrix multiply and the speed at which it can do string sorting. That's essentially the two routines that all these functions boil down to, is sparse matrix multiply and string storing. And those are essentially hardware limited. Those are very optimized functions.

So this just gives you a way. I think it's always-- the first thing we do when we run a program is if we want to optimize it, we'll write the program and run it, get a time, and then we'll try and figure out based on this type of data, well, what's the theoretical best we could do? Given what this calculation is dominated by, what is the best we can do?

Because if the best we could do-- the theoretical peak for what we were doing was 1,000 times better than what we're doing, that tells us, you know what? If we have to invest time in optimization, it probably is time well spent. Going from being 1,000x below whatever the best you can do getting to 100x usually doesn't take too much trouble. Usually, you can find what that issue is and you can correct it.

Likewise, going from 100x to 10x is still usually a worthwhile thing to do. If you're at 10% of the best that you can do, then you begin to think hard about, well, do I really want to chase? I'm never going to probably do better than 50% of what the best is, or maybe even 30% or 40%. Usually, there's a lot of effort that goes into going from 10% of peak performance to 50% of performance. But going from 0.1% of peak performance to 1% of peak performance is usually profile the code, aha, I'm doing something wrong here, fix it, done.

And so this is something that's a part of our philosophy in doing optimization, and why we spend so much time running benchmarks and understanding where-- because it tells us when to stop. If someone says, well, will this be faster, and you can say no, it won't be faster, you're done. Or, ah, no, we think in a few weeks' time, we can really make it a lot better. So very useful thing, and that's why benchmarking is a big part of what we do.

So finally, I want to wrap up here. This was one of the first charts I showed you, and hopefully now it makes a lot more sense. This shows you the easy path toward solving your problems in signal processing on databases. And if you look at your problem as one of how much data I want to process and how much-- what the granularity of access is, well, if I don't have a lot of data and I'm accessing it at a very small granularity, well, then I should just do it in the memory of my computer. Don't mess around with files. Don't mess around with databases. Keep your life simple. Work it there.

If the data request gets high, memory is still the best. Whether you're doing little bits or big bits, working out of main memory on a single processor, generally, for the most part, is where you want to be. If you're going to be getting to larger amounts of data but having a fairly large request size, then using files, reading those files in and out-- so basically, if I have a lot of data

I want to process, it won't fit into serial memory, we'll write it out into a bunch of files, just read them in, process them, move onto the next file is the right thing to do. Or spread it out in parallel. Run a bunch of nodes and use the RAM in each one of those. That's still the best way to do it.

You could also do that in parallel. You can have parallel programs reading parallel files, too. So if you have really enormous amounts of data and you're going to be reading the data in large chunks, then you want to use parallel files, parallel computing. That's going to give you better performance. And it's just going to be easy. Have a lot of files. Each program reads into sets of files, processes its set. Very easy to do. And we have lots of support in our [INAUDIBLE] system for doing those types of problems. It's the thing that most people do.

However, if you have a large amount of data and you want to be accessing small fractions of it randomly, that's when you want to use the database. That is the use case for the database. And there are definitely times when we have that case where we want to do that. But you want to build towards it. A lot of times when you're doing analytics or algorithm development, you tend to be in these cases first. And so rather than bite off the complexity associated with dealing with a database, use these tools first. But then eventually, there may be a time where you actually have to use the database. And hopefully, one thing you get out of this course is understanding this nuance type of thing.

Because a lot of people say, we need a parallel database. Why? We have a lot of data. All right. That's great. Well, what do you want to do? We want to scan over the whole thing. Well, that's just going to be a lot of data really slow. So people will do that. They'll load a bunch of data in the system. They're like, well, but nothing is faster. It's like, well, yeah, if you're going to scan over a significant fraction the database, it will be slower-- certainly slower than loading it into the memories of a bunch of computers, and even slower than just having a bunch of files and reading them into memory.

And increasingly nowadays, you're talking about this is millions of entries. There's a lot of things that were databases that we have, like the lab's LDAP server. It's like 30,000 entries. We have a database for it. You could send it around an Excel spreadsheet and pretty much do everything with that. Or the entire release review query, or the phone book, or whatever-- these are all now, by today's standards, microscopic data sets that can trivially be stored in a single associative array in something like this and manipulated to your heart's content. So there's a lot of data sets that are big to the human but are microscopic to the technology

nowadays.

So this is usually millions of entries. This might be hundreds of millions of entries. Definitely, when you really need-- often, you're talking about billions of entries when you're getting here to the point where you really need a database. The exception to this being if you're in a program that requires-- many of our customers will say the database is being used. This is where the data is. It's in no other place. So for integration purposes, then you throw all this out the window.

If they say, look, this is where the data is and that's the only way you can get it, then of course. But that still doesn't mean you might be like, yeah, well, you're going to store it in the database, but I'm going to store it in /temp, and work on it, and then put my results back in the database, because I know that what I'm doing is going to be better. There's no-- as they say, we don't want to be too proud to win.

Just because some people, it's like, we have a database, and everything must be purely done in the database. Or some people are like, we have a file system, and everything must be purely done in the file system. Or we have this language and everything-- the right tool for the job, pulling them together, totally fine. It's a quick way to get this work done without driving yourself crazy.

So with that, I'll come to the end. I have one code example which is relatively short to show you. But again, parallel graphs are the dominant type of data we see. Graph500 relies on this Kronecker graphs. Kronecker graph theory is a great theoretical framework for looking at stuff. You get all the identity benefits that come with Kronecker product eigenvalues. If you're into that, I certainly suggest you read the Van Loan paper, which is also the seminal work on Kronecker products. And I don't think anyone has written a followup to him, because everyone is like, yep, covered it there. It's about 12, 15 pages. Covered all the properties of Kronecker products.

We can do parallel computing in D4M via pMATLAB. And again, fundamentally, if you've implemented your algorithms correctly, they're limited by hardware, by fundamental aspects of the hardware. There are limitations. It's important to know what those are and to be aware of them.