JEREMY KEPNER: All right, well, I want to thank you all for coming to what I have advertised as the penultimate course in this lecture series. Everything else we've done up to this point has sort of been building up to actually finally really using databases. And hopefully, you haven't been too disappointed at how long I've led you along here to get to this point.

But the point being that if you have certain abstract concepts in your mind, that once we get to the database part, it just feels very straightforward. If we do the database piece and you don't have those concepts, then you can easily get distracted by extraneous information. So today, there's no view graphs. So I'm sure you're all thrilled about that.

It's going to be all demos showing interacting with the actual technologies that we have here. And everything I'm showing is stuff that you can use. I mean, so everyone has been-- so I'm just going to kind of get into this.

And let's just start with step one. We're going to be using the Accumulo databases that we've set up. We have a clearinghouse of these databases on our LLGrid system. And you can get to that list by going to this web page here, dbstatusllgrid.ll.mit.edu.

And when you go there, it will prompt you for your one password. And then it will show you the databases that you have access to. Now, I have access to all the databases. But you guys should only see these class databases if you log into that.

And so as you see here, we have five databases that are set up. These are five independent instances of Accumulo. And I started a couple already. And we can even take a look at them.

So this is what running Accumulo instance looks like. This is its main page here. And it shows you how much disk is used, and the number of tables, and all that type of stuff.

And it gives you a nice history that shows ingest rate over the last few [INAUDIBLE], and scan rate. This is all in entries per second, ingest in megabytes, all different kinds of really useful information here. And you'll see that this has got the URL of classdb01.cloud.llgrid. When I

started it, it was an actual machine that was allocated to that.

In fact, just for fun here, I could turn one of these on. You guys are free to start them. I wouldn't encourage you to hit the Stop button, because if someone else is using it, and you hit stop, then that may not be something you want to do.

But it's the same set up-- for instance, if you have a project, everyone in the class can see this because we've made you all a part of the class group. But you can see here there's other classes.

We have a bioinformatics group. And they have a couple of databases. Those are there. They're not running right now. There's a very large graph database group. It's running now.

And just to show this is it's running. You see this has about 200 gigabytes of data. And if we look in the tables here, we see here we have a few tables. Here are some tables with a few billion entries that have been put in there.

And this is really what Accumulo does very, very well. But I'm going to start one just for fun here, if that works. And so it will be starting that. And all that happens. And you can see it's starting, and all those type of stuff.

So we're going to get going here now with the specific examples. And I have these. Just so you know, today's examples-- so it's in the Examples directory, in the Scaling directory, and two parallel database. So this is the directory we're going to be going through today.

And we have a lot of examples to get through, because we're going to be covering a lot of ground here about how you can take advantage of D4M and Accumulo together here. So the first thing I'm going to do is go here. I'm going to run these.

I have, essentially, two versions of the code-- one that's going to do fairly smaller databases on my laptop. And then I have another version that's sitting in my LLGrid account that I can do some bigger things with. So to get started, we're going to do this first example PDB01 data test.

So in order to do database work, and to test data, we need to generate some data. And so I'm using a built-in data generator that we have called the Kronecker graph. It's basically borrowed from a benchmark called the Graph 500 benchmark. There's actually a list called Graph 500.

And I helped write that benchmark. And, in fact, I actually-- the Matlab code on that website is stuff that I originally wrote, and other people have since modified. And so this is a graph generator.

It generates a very large power law graph using a Kronecker product approach. And it has a few parameters here-- a scale parameter, which is basically the number of vertices. So 2 this scale parameter is approximately the number of vertices. So 2 to the 12th gets you about 4,000 vertices.

It then creates a certain number of edges per vertex-- so 16 edges per vertex. And so this computes n max as 2 to the scale here. And then the number of edges is edges per vertex times n max. This is the maximum number of edges.

And then it generates this. And it comes back with two vectors, which is just the list of the-- the first vector is a list of starting vertices. And the second vector is a list of ending vertices. And we're not really using any D4M here. We're just creating a sparse adjacency matrix of that data, showing it, and then plotting the degree distribution.

So if we look at that figure-- so this shows the adjacency matrix of this graph, start vertex to end vertex. These Kronecker graphs have this sort of recursive structure. And if you kept zooming in, you would see that the graph looked like itself in a recursive way here.

That's what gives us this power law distribution. And this is a relatively small graph. This particular data generator is chosen because you can make enormous graphs in parallel very easily, which is something that if we had to pass around large data sets every single time we wanted to test our software, it would be prohibitive because we'd be passing around gigabytes and terabytes. And I think the largest this has ever been run on is on a scale of 2 to the 37. So that's trillions of vertices, or certainly billions of vertices, almost trillions of vertices.

And then we do the degree distribution of this. And you see here, it creates a power law distribution. We have a few vertices with only one connection. And we always have a super node with a lot of connections.

And you can see, actually, here the Kronecker structure in this data, which creates this characteristic sawtooth pattern. And there's ways to get rid of that if you want. But for our purposes, having that structure there, there's no problem with that. So this is kind of exactly what the degree distribution looks like. So that's just a small version to show you what the data

looks like. Now we're going to create a bigger version.

So this program, which I'll now show you-- so this program creates, essentially, the same Kronecker graph. But it's going to do it eight times. And one of the nice things about this generator is if you just keep calling this, it gives you more independent samples from the same graph. So we're just creating a graph that's got eight times as many edges as the previous one just by calling it over and over again, just from the random number generator.

So I have eight. I'm going to do this eight times. And I'm going to save each one of those to a separate file. So I create a file name. I'm actually setting the random number seed to be set by the file name so that I can do this if I want to-- the seventh file will always have, essentially, the same random sequence regardless of when I run it.

And so I create my vertices. And I'm going to convert these to strings, and then write these out to files. And so that's all that does here.

And one of the things I do throughout this process that you will see is I keep track of how many edges per second I'm generating things. So here, I'm generating about 150,000. It varies in terms of the edges per second here, but between 30,0000 and 150,000 or 180,000 edges per second, because when you're creating a whole data processing pipeline, that's essentially the kind of metrics you're looking-- some steps might process your edges extremely quickly. And other steps might process your edges more slowly. And that's, obviously, the ones where you want to put more energy and effort into it.

So we can actually now go and look. It stuck it in this data directory here. And we just created that. And so, basically, we write it out on three files. Essentially, each one of these holds one part of a triple-- so a row, a column, and a value.

So if we look at the tow, you can just see it's a sequence of strings separated by commas, same with the column, just a separate sequence of strings separated by commas. And then in this case, the values we just made all ones, nothing fancy there.

So now we have eight files. That's great. We generated those very quickly. And now we want to do a little processing on them.

So if we go pDB03, the first thing we're going to do is read those files back in and construct associative arrays, because the associative ray construction time takes a little time. And we're going to want to use it over and over again. So we might as well take those triples and

construct them into associative arrays, and save them out as malak binary files. And now that will be something that we can work with very quickly.

So we're going to do that. So there you go. It read them in. And it shows you at the rate at which it reads them in, and then essentially writes them out, and then gives us another example of the edges per second.

And now you see we have Matlab files for each one of those. And, not surprisingly, the Matlab file is smaller than the three input triples that it gave. So this is a 24 kilobyte Matlab file. And it was probably about 80 kilobytes of input data.

And that's just because we've compressed all the row keys into single vectors. And we have the sparse adjacency matrix, which stores things. And so that makes it a little bit better there. If we actually look at that program here-- so we can see we basically are reading it in.

And then what we're doing is we're basically creating an associative array. We read in each set of triples. And then the constructor takes the list of row strings, column strings. We just all want this-- since we knew they were all one, we were just letting that be one.

And then there's this optional fourth argument that tells us, what do we want to do if we put in two triples with the same row and column, what to do? The default is it will just do the min. So if I have a collision, it will just do a min.

If I give it this optional fourth argument at sum-- in fact, you can put in essentially any binary operation there. But at sum will just add them together. So now we'll have-- in the associative array, a particular row and column will have how many that occurred.

And so we're summing up as we go here. And then after we create the associative array, we save them out to a file. And so we have that step done.

Now, the whole reason I showed you this process is because now I'm actually in a position I can start doing computation just on the files. As I said before, I don't have to use the database. If I'm going to do any kind of calculation that's going to involve traversing all the data, it's going to be faster just to read in those Matlab files and do my processing on that.

It's also very easy to make parallel. I have a lot of files. I just have different-- if I launch a parallel job, I can just have different processes reading separate files. It will scale very well.

The data array read rates will be very fast. Reading these files in parallel will take much less time than trying to pull out all the data from the database again. So we're going to do a little analytics here. pDB04 so I'm going to basically compute-- I'm going to take those eight files, read them all in, and accumulate the results as we go.

And there we go. We get to the in degree distribution and the out degree distribution of this result. If you look to that program here, you can see all we did is we looped over all the files, just loaded them from in Matlab, and then basically summed the rows and added that to a temp variable, and summed the columns, and then plotted them out. So we just sort of accumulated them as we went.

This actually-- this method of just summing on top of an associative array is something that you can certainly do. It's a very convenient way to do it. I should say, though-- and you can kind of see it here a little bit. You notice that the time is beginning to-- it's not so clear here, this took so little time.

But on a larger example, what you would see is that every single time we did that-- because we're building and then adding, we're basically redoing the construction process. And so, eventually, this will become longer, and longer, and longer. And so it's OK for small stuff to do that, or if you're only going to do it a few times.

But if you're going to be accumulating an enormous amount of data, then what we can actually do is we have another version of this program pDB04 cat DegreeTest And you can tell that was a little bit faster. You see here it's all in milliseconds of time.

And this is a little bit longer program. What we're doing here is, basically, we're reading in-- doing the exact same thing. We're loading our Matlab file. We're doing the sum.

And then since I know something about the structure of that. That is, basically I'm summing the rows. I can just append that to a longer list, and then at the and do one large sum. And that's, obviously, much faster.

And so these kind of tricks you just need to be aware of. If you're trying to do very-- people typically want to do a large amount of data. You just do the simple sum. That will be OK.

But if you're doing a lot over a large list, that essentially becomes almost an n squared operation with the loop variable. And this is one that will be even faster. You can make it even faster, because we are doing this concatenation here. When you do a concatenation in

Matlab, you're doing a malak.

If you want to make it even faster, you can pre-allocate a large buffer, and then append and into that buffer. And then when you hit the edge, do a sum then, and do it that way. And that's the fastest you can do.

So these are tricks-- very, very large sums, you can do them very quickly, and all with files. You don't need a database. And this is the way to go if you're going to be doing an analytic where you really want to traverse most of the data in the database, in your data set. If you just need to get pieces of the data in the database, then the database will be a better tool.

We did that. So those show how we worked with files. And that's always a good place to start.

Even if you are working with the database, if you find that you're doing one query over and over again that you're going to be then working with that data, a lot times better to just do that query, and then save those results to a file, and then just work with that file while you're doing it. And, again, this is something that people often do in our business.

Now we're going to get to the actual database part of it. So the first thing we're going to do is we have to set up our database. So we're going to create some tables in Accumulo.

And so we want to create those first so they're created properly. And so I'm going to show you the program that does that. The first thing that you're going to do, it's going to call a DB setup command. And so let me-- this program, I'm going to now show you that. And so when you run these examples, you will have to modify this DB setup program.

So the first thing you'll notice is that we're all using the same-- each group that's with those databases is all using one user account. And you can say, well, that's not the best way to do it. Well, it's very consistent with the group structure, in that it's basically you're all users. The database is there to share data amongst your group. And so it is not an uncommon practice to have a single user account in which you put that data.

So we have a bit of a namespace problem. If you all just ran this example together, you'd all create the exact same table, and all fill it up. So the first thing we're going to do is just pre-pend the tables. And I would suggest that you put your name-- instead of having my name there, you put your name there.

And then we have a special command you're called DB Setup LLGrid, which basically creates

a binding to a database just using the name of the database. So it's a special function. That's not a generic function. It only works with our LLGrid system. And it only works if you have mounted the LLGrid file system.

So hide all this stuff here, get rid of that. So as you see here, I have mounted the yellow grid file system. And you need to do that because the DB setup command, when it binds to the database it actually goes get the keys from the LLGrid file system.

And those keys are sitting in the group directory for that. So, basically, from a password management perspective, all we need to do is add you to the group. And then you have access to the database.

Or if we remove you from the group, you no longer have access to the database. So that's how we do-- otherwise, we'd have to distribute keys to every single user all the time. And so this is why we do that.

So this greatly simplifies it. But you will not be able to make a connection to one of these databases unless you are either logged into your LLGrid account. Or if you are on your computer, you've mounted the file system, and D4M knows where to look for the keys when you pass in that setup command.

So if we look at that again-- and this is just a shorthand for the full DB server command. So if you were connecting to some database directly other than one of these-- or you could even do it with these-- you would have to pass in, essentially, a five argument thing, which is the hostname of the computer and the port, the name of the instance, the name of the-- I guess there's a couple instance names here-- and then the name of the user, and then an actual password. And so that's the generic way to connect in general. But for of those of you connected to LLGrid, we can just use this shorthand, which is very nice.

Then we're going to build a couple of tables. So we have these-- first thing we're going to do is we're going to want to create a table that's going to hold that adjacency matrix that I just created with the files. And so we're going to do that with a database pair.

So if we have our database object here and we give it two string names, it will know to create two tables in the database, and return a binding to that table that's a transposed pair. So whenever we do an insert into that table, it will insert the row and the column in one table, and then flip those and insert the column and the row in the other table. And then whenever you do

a lookup, it will know if it's doing a row lookup to look on one table, and if it's doing a column lookup to do it on the other table. And this allows you to do fast lookups of both rows and columns and makes it all nice for you.

We're also going to want to take advantage of Accumulo's built-in ability to sum as we insert. And so we're going to create something that's going to hold the degree as we go-- very useful to have these statistics, because a lot of times you want to look up something. But the first thing you want to do is to see, well, how many of them are in there? And so if you create, essentially, a column vector with that information, it's very helpful.

Later, we're going to do something where we actually store the raw edges that were created. So when we create the adjacency matrix, we actually lose a little bit of information. And so when we create this edge matrix, we'll be able to preserve that matrix, that information. And, likewise, we'll be doing the tallies of the edges in that as well.

So that's what this does. And that's the setup. You'll need to modify that and this program here.

Actually, basically after the setup is done, it adds these accumulator things by designating certain columns to be what they call combiners. So in this adjacency degree table, I've said, I want to create two new columns-- an out degree, in degree column, and the operation-- that I want to be applied when there are collisions on those values is sum. It will then sum the values. And, likewise, with the edge, since I only have one I'm going to have to degree and sum there. So that's how we do that.

So if we now go to our database page here-- so you can see, class DB3, it started. We can actually view the info. And this is what a nice, fresh, never before used Accumulo instance looks like. It has very little data.

It has one table, a meta data table and a trace table. And there's no errors or anything like that-- so a very clean instance. So that's what one looks like.

We're going to use one that we've already started though, which is DB1. And if we look at the tables here, there's already some tables. Michelle ran a practice run on this. [? Chansup ?] ran a practice run.

I'm going to now create those tables by live setup test. There, it created all those tables. You can now see, did it actually work?

So if I refresh that-- and you can see, it's now created these six tables which are empty. We do have abilities to set the write and read permissions of these tables. So right now, everyone has the ability to read, and write, and delete everyone else's tables in a class database.

In a project, that's not such a difficult thing to manage. You all know that. But you could imagine in a situation [INAUDIBLE], we had a big ingest. This is a corpus of data. We don't want anybody-- we can actually make the permissions. We can make it read only so that no one can delete it. Or we can make it so it's still read and write, but it can't be deleted, whole cloth, those permissions exist.

A feature we will add to this database manager will also be a checkpoint feature. So, for instance, if you did a big ingest, have a bunch of data that you're very happy with, you can checkpoint it. You'll have to stop the database. Then you can create a checkpoint of that stop database, name checkpoint. And then you can restart from that, if for some reason your database gets corrupted.

As I like to say, Accumulo, like all other databases, is stable for production. But it can be unstable for development. New database users, the database will train you in terms of the things that you should not do to it.

And so over time, you will not do things that destroy data, or cause your database to be very unhappy. And then you will have a nice production database, because you will only do things that make it happy. But in that phase where you're learning, or you're experimenting with things, as with all databases, any database, it's easy to do commands that will put the database in a fairly unhappy state, even to the point of corrupting or losing the data.

But for the most part, once it's up and running in production-- and we have a database that's been running for almost two years continuously using a very old version of this software. So it just continues to hum away. It's got billions of entries in it. It's running on a single Mac. And so it definitely, we've seen situations where it's-- it has the same stability as just about any other database.

So now we've created these tables. And let's insert some data into them. So I do pDB06. So I'm now going to insert the adjacency matrix.

And now it's basically ereading in each file and ingesting that data. And it's not a lot of data. It

doesn't take very long.

And now you can see up here that data is getting ingested. And you see there. So we just inserted about 62,000 in each of those two tables, and 25,000, which for Accumulo is a trivial amount.

We just inserted 150,000 entries into a database, which is pretty impressive, to be able to do that in, essentially, the blink of an eye. And that's really the real power of Accumulo, is this-- on a lot of other databases, 150,000 entries, you're talking about a few minutes. And here, it's just you wouldn't even think about doing that twice.

So we can take a look at that program. So here we go. So we had eight files here, and basically loaded the data. And, basically, one thing we have to remember is that our adjacency matrix has numeric values in an associative array.

And Accumulo can only hold strings. So we have to call NumtoString function, which will convert those numbers into strings to be stored. So the first thing we do is we load our adjacency matrix A.

We convert the numeric values to strings. And we just do a put. So we can just insert the associative array directly into the adjacency matrix.

It pulls apart the triples. And it knows how to take care of the fact that it recognizes this is a transposed table pair. And it does that ingest for you.

And, likewise, same thing here-- now on these other things, we pulled it out. We summed it, convert it to strings to do the out degree, and the same thing to do in degree. And so this is actually where the adjacency matrix comes in very handy because when we're doing accumulating, if we didn't first sum it and then do it-- if we put those raw triples into insert, we're essentially redoing the complete insert. And so this saves usually an order of magnitude in number of inserts by basically doing the sum in our D4M program first, and then just inserting those sum values, just a nice way to save the database a little bit of trouble in doing that. And so we certainly recommend this type of approach for doing that.

So the next one DB07. So now we're going to a query. We're going to get some data out of that table.

And so what we did here is we said, I want to pick 100 random vertices. So in this case, I

randomly generate 100 random vertices over the range 1 to 1,000. OK And I then convert those to a string, because these are numeric values. And I will look up strings.

And then the first thing I'm going to do is I'm going to look up the degrees, essentially, of those vertices. So I have my sum table here called T adjacency degree. I'm going to pass those strings in.

And then I'm going to get just the degrees. So that's just a big, long, skinny vector. Looking things up from that takes much less time than looking up whole rows or columns. And it stores all those values for me. So that's a great place to start.

And then I want to say, you know what? I only care about degrees that have been a particular range. This is a very common thing to do.

There will be vertices that are so common you're like, I don't care about those. I have their tally. And these are sometimes called super nodes. And doing anything with those is a waste of my time, and forces me to end up traversing enormous swathes of the database.

So I'll set an upper threshold here and a lower threshold. So, basically, I take a degree. I'm just going to look at the out degree. And I say, I want things that are greater than the min and less than the max. And I want to get just those rows.

So that's this query, a fairly complicated thing, analytic here done in just one line. And now a new set of vertices, which are just the rows that satisfy this requirement. And then I'm going to look those up again. So I'm just going to look up the ones that satisfy those-- now I'm going to get the whole row of those things. So before, I was just looking up their counts.

Now I'm going to get the whole row. And I know that there's none that have more than this certain value, or have too few. And then I'm going to plot it.

And so if we look at the figure, there we see. So we ended up finding one, two, three, four rows that had between-- these should all write-- one, two, three, four, five, six, seven, so that's between five and 10. These should all be between five and 10.

And then this shows what their actual column was. And so that's a very quick example of that kind of analytic. And, again, real bread and butter, and this is basically standard from a signal processing perspective.

The max is our clutter threshold. We don't want that. It's too bright. And then we'll have a noise threshold. We're like, we don't care anything that's below a certain value.

And this sort of narrows in on the signal. Same kind of processing and signal processing, we're doing it here on graphs. And Accumulo supports that very, very nicely for us.

So let's move on to the next example. And, actually, if we look here, if we go back to the overview-- so you see here, that was that ingest I did. This is the ingest-- so very quick, it's getting about 5,000 inserts a second.

That was over a very short period of time. It doesn't even time to get reach its full rise time there. And then you can see the scan rate.

And it was very small. It was a very tiny amount type of thing. As we do larger data sets, you'll really see that.

And this is a great tool here. It really shows you what's really going on. Before you use the database, always check this page.

If you can't get to the page of the database, you're not going to be able to get to the database. There's no point in doing anything with D4M if this page is not responding. Likewise, if you look at this, and you see this thing is just hamming away. You're seeing hundreds of thousands or millions of inserts a second, it means that, guess what? Someone is probably using your database. And you might want to either pick a different database, or find out who that is and say, hey, when are you going to be done, or something like that? Likewise, on the scan side.

Likewise, when you do inserts-- you'll get some examples here of the kinds of rates you should be seeing. You want to make sure you're seeing those rates. If you're not seeing those rates, if you're basically just using the resource but only inserting at a low rate, then you're actually doing yourself and everybody else a disservice.

You're using the database. But you're using it inefficiently. And it's much better to have your inserts go fast, because then you're out of the way.

Your work gets done quicker. And then you're out of everybody else's way quicker too. So I highly recommend people look at this to see what's going on.

So I think the last one was seven. So we'll move on to eight here. So now we're going to do

another type of query. We're going to do a little bit more sophisticated query

That query used the degree tables to sort of prune our query. So you think about it. There was probably an edge in there that had like 100 entries.

And we just were able to avoid that, never had to touch that edge. But if we had touched it, that could have been a much bigger query. You might be like, well, 100 doesn't sound bad. Well, it's easy to get databases where you'll have some rows with a million entries, or columns with a million entries, or a billion entries. And you really don't want to have to query those rows or columns, because they will just send back so much data.

But you can still have situations where you're doing queries that are going to take-- they're going to send back a lot of data, more data that you can really handle in your memory segment. So we have a thing called an iterator. So we've created an iterator log to set up a query and have it work through it.

Now, this table is so small that you won't really get to see the iteration happening. But I'll show you the setup. So we'll do that. So that was very quick.

So, essentially, we're doing a similar thing. We're creating a random set of vertices here, about 100, creating over the range 1 to 1,000. And we're creating an iterator here called Tadjacency iterator. It's this function here, iterator.

We pass it in the table. Then we have a flag, which is element, which is, how many entries do we want this iterator-- it's the chunk size. How many entries do we want this iterator to return every single time we call it?

And here, we've set max element to be a pretty small number, to be 1,000. So what we're saying is every single time we do this query, we want you to return 1,000 at a time. Now for those of you who are Matlab aficionados, you should be in awe, because Matlab is supposed to be a stateless language.

And there's nothing more stateful than an iterator. And so we have tricked Matlab with a combination of Matlab on the surface and some hidden Java under the covers to make it have the feel of Matlab, yet hold state. So now what we're going to do is we're going to-- this just creates the iterator.

We then initialize the query by actually passing in a query. So we now say, OK, the query we

want is over this set of rows here. And so we're going to run the query the first time. And since that thing returns string values, and we want numbers, we have to do a string to num.

And that's our first associative array that's the result of the first query. We then initialize our tally here. And then we just do a while on this result.

So we're going to say, oh, if there's something there, then we want sum that, and add it to our tally, to our in degree tally. And then we call it again. So we do the next round of the iteration by just calling the object with no argument.

And it will just run it until the query is empty. If we put in an argument, it would re-initialize the iterator to that new query. And so you don't have to create a new iterator every single time you want to put in a different query.

You can reuse the object. Not that it really matters, but this is how you get it to do again. So it's a pretty elegant syntax for basically doing an iterator. And it allows you to deal with things very nicely.

And as we see here, then we did the calculation here, which is all right. I want it to then return the value with the largest maximum degree. So, basically, I compute the max. I get the adjacency matrix of the degree. And I compute its maximum.

And then I set that equal to n degree. And it tells me that the first vertex had a degree of 14 in this query, which makes sense. In the Kronecker matrix, 1, 1 is always the largest and densest value. So that's just how we use iterators. Let us continue on here.

Now I'm going to use iterators in a more complicated way to do a join. So a join is where I want to basically-- maybe there's a row in the table. And I only want the rows that have both of a certain type of value in them.

So, for instance, if I had a table of network records, I might like, look, please only return records that have this source IP, and are talking to this domain name. So we have to figure out how we do joins in this technology. So I'm going to do that. So let me run that.

So join-- so a little bit more complicated, obviously, we're building up fairly complicated analytics here. So I create my iterator limit here. I'm going to pick two columns to join, column 1 and column 2.

And this just does a simple join over those columns. So basically what I'm doing is I'm saying, please return all the columns that contain-- this is an or basically either column 1 or 2. I'm going to then convert those values, which would have been string values of numerics, to just zeros or 1. Then I'm going to sum that.

So, basically, I got the two columns. I converted all the values to 1. Now I'm going to sum them together.

And then I'm going to ask the question, where were those equal to 2? Those show me the records. So that's what I'm doing here. I'm saying, those equal to 2.

So that shows me all the records where that value is equal to 2. And I can then get the row of those things, and then pass that back in to the original matrix. And now I will get back those rows with those things.

And so we can see that. I think that's the first figure that we did here. We go to figure one. And those show, basically-- what did we do? Right.

This shows us the complete rows of all the records that had the value 1. So this is here, 1, and I think it was 100, which is somewhere right probably in there. So basically every single one of these records had 1 and 100 in it.

And then it shows us all the rest of the values that are also in that record. If we just wanted those columns and those values, when we just summed them equal to 2, we were done. But this allowed us to then go back into the record, and now into the full set, and just get those records. So this is a way to do a join if you can hold the complete results of both of those things, like give me the whole column 1, and the whole column 2. That's one way to do a join, perfectly reasonable way to do a join.

I'm going to now do this again, but with a column range. So I'm going to say, I want to do a join of all columns that begin with 111, and all columns that begin with 222. So that would return more.

I'm going to create two iterators now to do that. So I have initialized my iterator, iterator one and iterator two. And so now I'm going to start the first query iterator here by giving it its column range that initializes it.

And I get an A1. And then I check to see if A1 is not empty. If it isn't empty, I'm going to then

sum it, and then call it again until it proceeds. And since it's such a small thing, it only went through that once.

And then now I'm going to do the same thing again with the other iterator and sum them to get together again. And now I'm going to join those two columns. And that's a way of doing the join with essentially two nested iterators, and doing the join that way. So that's something you can do if you couldn't hold the whole column in memory and you wanted to build it up as you went. That's a way to do it with iterators.

Then let's see here. There's an example of the results from that. And just so you know, when you do an SQL query in an SQL database, this is what it's doing under the covers. It's trying to do whatever information it can.

It will [INAUDIBLE] hold a lot of internal statistical information. It's trying to figure out, can I hold the results in memory? Or can I not? Do I have to go through in chunks? Or do I not?

Do I have information about oh-- you know, this query, do I have a sum table sitting around that will tell me, oh, which column should I query first, because it will return fewer results? And then I can go from there type of thing. So this is all going under the covers.

Here, you get the power to do that directly on the database. And it's pretty easy to do. But you do have to understand these concepts. So now we'll move on.

So that was all with the adjacency matrix. And as I said before, when we formed the adjacency matrix, we threw away a little information, because if we had multiple-- if we had a collision of any kind, we lost the distinctness of that record, or that edge. And a lot of times, like, no we want to keep these edges, because, yeah, we'll have other information about those edges. And we want to keep things.

So we want to store that. So we're going to now reinsert the data in the table, and use, essentially, instead of an adjacency matrix, an incidence matrix. An incidence matrix, every single row is an edge. And every single column is a vertice.

And so an edge can then connect multiple vertices. It also allows us to store essentially what we call hyper edges. So if you have an edge that connects multiple vertices at the same time, we can do that. So let's do that.

This is inserting about twice as much data. So it naturally takes a little bit longer there. And you

see the edge insertion rates that we're getting there, 30,000 edges per second. So let's go and see what it did to our data here.

So if we look at our tables, we can see now that there's our edge data set. And you see we've inserted about 270,000 distinct entries in this data. So there's the edge table. And there's this transpose.

And there's the degree count. And as you saw before, we had 53,000. So that just shows you the additional information.

Let's look at that program here. So, again, we're looping over all our files here. We're reading them in.

I should say, this case we're reading in the raw text files again. We're not reading in the associative array because we just want to insert those edges. And then the only thing we've done is that, basically, to create our edge we had to create-- this data didn't come with a record label.

So we don't have any. So we're constructing a unique record label for each edge here just so that we have it for the row key. And then we're pre-pending the word out into the row string and in into the column string.

So we know when we create our record, out shows the direction it came from. And in shows the direction it left. And so that's a way of creating the edge. And then, likewise, we do the count degree and such to preserve that information so we can sum the new total number of counts there.

So let's do some queries on that. So I'm going to ask for 100 random vertices here. So I get my random vertices.

And then I'm going to do my query of the strings. But I have to pre-pend this out, slash, the value in it so I know I'm looking for vertices from which the edge is departing. And I'm going to get those edges.

So I get those edges. So I created the query. I get the edges. I'm going to do my thresholding again.

I want a certain min and max. And then I'm going to do the threshold. So this just gave me the

degree counts.

And I thresholded between this range. And then now I then do the same thing back with the-- I say, give me everything greater than degree min and less than degree max. I get a new set of rows.

So that will just return the edges that are a part of vertices with these degree range. And then I'm going to get all those edges, all the records that contain those vertices, through this nested query here. The result is this.

So, basically, this shows me all the edges there are a part of this random set of vertices that have a degree range between five and 10. This is a fairly sophisticated analytic. We're doing about seven or eight queries here, doing a lot of mathematics.

And you see how dense it is. And, hopefully, from what we've learned in prior has some intuition for you. And we'll continue on.

So now I'm going to do a query with the iterator-- again, same type of drill. I set the maximum number of elements to the iterator. I get my random set of things. I create an iterator, again setting the number of elements.

I initialize the iterator to be over these vertices. I then check to see if it returned anything. If it did, I'm going to then actually pass the rows of that back into it to get the edges containing those vertices, and then do a sum to compute the in degree, and so on and so forth. And then I get here the maximum in degree of that set of vertices was 25.

So that's just an example of that. And that was 12. And I think 13 is our last one.

All right, and again, a more complicated example showing basically a join over this space creating, essentially, a couple of sets of edges, a couple of column ranges, iterators, and so on and so forth. And I won't belabor this point. But this just shows how you can combine between using the degree table and iterators.

You have all the tools at your disposal that any type of query planning system would have inside it, that it would use to make sure that you're not over-taxing the results that you're returning too. And that's a lot of times if you do a query on any database, you get the big spinning watch or whatever. It's because the query you asked was simply too long.

It also gives a lot of nice places to-- if you're making a query system, to interrupt it. So if you do the query against the counts, you can quickly tell the user, look, you just did a query. And this is going to return 10 million results. Do you want to proceed? And so you, of course, [INAUDIBLE].

Or likewise, you can set a maximum number of iterations. Like it says, OK, I want to get them back in units of 100,000 entries. But I only want to go up to a million. And then I'm going to pause and get some kind of additional guidance from the user to continue. So those are the same tools and tricks that are in any query planner very elegantly exposed to you here for managing these types of queries.

With that, I want to do some stuff where we do a little bit of bigger data sets. So I've walked through the examples. I want to show you what this is like running on a bigger data. So let's close all that. Close that.

I want to do this one. So now I'm logged into-- I just SSHed into classdb02.cloud.llgrid.ll.mit.edu, which happens-- it tells you which node it's actually mapped to, which is node F-15-11 in our cluster. And this is a fairly powerful compute node.

These are our next generation compute nodes for LLGrid. So those of you who've been using LLGrid for all these past years, may have noticed that they're getting a little long in the tooth. These are the first set of the new nodes. And we'll have about 500 of them total in various systems.

And so here, I am doing something a little bit larger. So let me see here. Examples-- so I'm on my LLGrid account here. And I'm going to go to 3 and then 2. Then I do open dots.

So that's the directory. And so the first thing I did is in my DB setup here, you'll notice that I have done class DB 0. And also, we don't need to do 1. But I will do 2 here.

I've made this bigger. So I have made this now 2 to the 18th vertices, instead of what I had before. So let's go [INAUDIBLE] anymore.

So if I do PDB02, it's going to now generate these things. And so it's generating a lot more data. And you see it's doing at about 200,000 vertices per second. Just shows you the difference between the capability of my laptop and one of these servers here.

And this will also-- I'm logged onto this system. It has 32 cores. I can do things in parallel.

And so, for instance, if I did eval p run, for those of you who have had the parallel Matlab training, I can say before. And since I'm logged into this node, and I just do curly brackets, it just says launch locally on that node. Don't launch onto the grid.

And now it's launching that in parallel on this node data one, did that. Data two, did that. Now it's done. And the others finished there work too, probably right about now. So that's just an example of being able to do things in parallel.

We've created here-- I mean, you look at it. We did eight times 300,000. We did 2.5 million edges in that, essentially, five seconds type of thing. So multiply this by 4. You're doing like a million edges a second just in that one type of calculation there.

Move on TBPB03. And-- oh, I should say, I did modify that program slightly. Let's see here. So if I look at-- the line labeled in big capital letters parallel, I uncommented it.

That's what allows each one of the processors when they launched to then work on different files. Otherwise, they all would have worked on the same files. So by uncommenting that parallel, this now becomes a parallel program that I can run with evalp run command.

Of course, you have to have parallel Matlab installed, which of course you all do since you're on LLGrid. But for anyone seeing this on the internet, they would need to have that software installed too, which is also available on the web and installable there. So that's all we needed to do, was uncomment that one line to make that program parallel, and did the right thing for us as well.

And we're going to go on to the next example. And we did the same thing there. We just uncommented parallel. And it's now going to create these associate arrays in parallel.

So if I do PDB30-- so it's now actually constructing these associate arrays. You see it takes about four seconds to do each one of those. It's doing about 120,000, 130,000 entries per second. So this thing will take about 25 seconds to do the whole thing.

And, again, if we ran that in parallel, it automatically tries to kill the last job you ran if you're in the same directory so that you're not [INAUDIBLE] on top of yourself. And now you see it's doing that again. And now it's done.

And the other one is finished as well. You can actually check that, if you really want to. Just hit Control Z, if I do more [INAUDIBLE] out. You can see those for the output files of each one of

them.

I'm not lying. They didn't take a ridiculous amount of time. They all completed. You always encourage people to check those dot out files, and then that [INAUDIBLE] directory. That's where it sends all the standard out from all the other nodes.

It's probably the number one feedback we get from a user who says, my job didn't run. We're like, what does it say in your .out files? And usually, like, oh there's an error on node 3 because this calculation is wrong on that particular node, or something like that-- so just reminding folks of that.

Moving on-- so what did we just do? We did three. So we did PDB4, re-test. And this is doing a little bit bigger calculation.

And so you can see here-- I told you it does begin to get bigger here. So it started out-- the first iteration was about 0.6 seconds. And then it goes on to 0.8 seconds. If we did that cat approach, it would do it faster.

I'll show you a little neat trick. This is also a parallel program when I run it. And, basically, I loop over each file here.

I'm just doing this little ag thing just to sync the processors just for fun so I don't have to wait for them to start. And then it's going to go. And they're going to do their sums.

And then when they're all done, they're going to call this-- so each one will have a local sum. And it needs to be pulled together. So we have this function called GAG, which basically takes associative rays and we'll just sum them all together, and very nice tool for doing that.

And, of course, we had to uncomment that in order for that to work. And so let's go give that a try. And so if we do eval pRUN, so it's launching them. And it's reading. And then now it's going to have to wait a second.

OK, so it took about two seconds to pull them all together and do the sum across those processors. So that's a parallel computation, a classic example of what people do with LLGrid and can do with D4M is they have a bunch files. Each processor processes them independently. And at the end, they pull something all together using this GAG command.

All, right moving on-- so now I'm on database 2 here. So I'm going to go to that. And let's look

at our tables.

Very little activity-- and you see we have no tables there. So I have to create them. So I'm going to pd [INAUDIBLE] setup 05. That's going to create those tables on that database.

We can now look, see. And it created all my tables. So now I'm ready to go.

And now we're going to be insert again, PDB06. I'm going to insert the adjacency matrix. And this obviously takes a little bit longer. Each one of these, there's a parameter associated with the table, which is-- you would think, normally, it should just send all its data to the database at once.

But it turns out one thing we found is that actually the database prefers to receive the data in a smaller increment, typically around half a megabyte chunk. So every single time it's calling this, it's sending half a megabyte waiting to get the all clear again, and then setting the next one.

And we've actually found that makes a fairly significant-- so you can see here, we're getting about 60,000 inserts per second. This is from one processor. And this takes a little while.

Let's see if we can go and look at it while it's going on. If we go here, we should begin to see some. So there you go.

That's what a real insert is beginning to look like. It just changes its axis for you dynamically here. But we're getting about 60,000 inserts a second there.

Let me just go along here. It'll start leveling off a little bit. And then it will show you what's going on in the tables there.

And, I mean, not too many of you are probably database aficionados. But 60,000 inserts a seconds on a single node database is pretty darn amazing. I mean, you would mostly have had to use a parallel database. And that's actually one of the great powers of Accumulo is there's a lot of-- even though it's a parallel database, there's a lot of problems you can now do with a single node database that you would have had to have a parallel system to do before.

And that's really-- because, parallel computing is a real pain. I should know. If you can make your parallel problem fast enough to now work on a single one, that's really a tremendously convenient capability.

So we inserted there, what, eight million entries-- so pretty impressive. But that did take a while. And so maybe I want to do that in parallel.

So if I just do eval pRUN, let's try four and see what happens. Now I would expect, actually, this to begin to top this thing out. And so the individual inserts rates on each node probably go down a little bit.

And you'll see it will get a little bit noisy here, because now we have four separate processes all doing inserts. You see there was one. It took a little bit, almost a second. And you get this noise beginning to happen here.

But we're getting 50,000 edges per second on one node, which means we should be getting close to four times that. So let's go check. What's it seeing here?

So if we update that-- and there you see, we're sort of now climbing the hill well over 100,000. That was our first insert there. And now we're entering the second one here. Whoops, don't want to check my email.

Let's see here. So how are we doing there? Oh, it's done. We may not have even get the full rise time of that. Yeah, so it basically finished before we could even really hit the-- it has a little filter here, has a certain resolution.

You really need to do an insert for about 10 minutes before you can get really a sense of that. But there you see, we probably got over 200,000 inserts a second using four processes on one node. And we could probably keep on going up this ramp.

For this data set, I'd expect we'd be able to get maybe 500,000 inserts a second if I kept adding processors and stuff like that. And if you look at our data, what do we got here? We got like 15 million entries now in our database.

Again, one of the nice things is for a typical databases, a lot of times if you have to re-ingest the whole database, that's fine. In our cyber program, we have a month of data. And we can re-ingest it in a couple hours.

And that's a very powerful tool to be able to like, oh, you know what? I didn't like the ingestion. That's fine. I'll just rewrite the ingestion and redo it. And this gives you a very powerful tool for exploring your data here.

So that's kind of what I want to do with that. One of our students here very generously gave me some Twitter data. And so I wanted to show you a little bit with that Twitter data, because it's probably maybe a hair more meaningful than this abstract Kronecker graph data.

And by definition, Twitter data is about the most public data that one can imagine. I think no one who posts to Twitter is expecting any sense of privacy there. So I think we can use that OK.

So let's see here. Let me exit out of that. [INAUDIBLE] desktop, [INAUDIBLE], Twitter.

And so, basically, just a few examples here-- the first thing we did is construct the associative array. So let's start up here. And I think it was like a million Twitter. Is that right? A million entries, and how many tweets do you think that was?

We should be able to find out, right? We should be able to find out. So let's do the first thing here. So it's reading these in, and chunked, and writing them out to associative arrays.

That's going to be annoying. Isn't it? Let's go to a faster system. So this is running on the database system.

And I broke it up into chunks of 10. I couldn't quite on my laptop fit the whole thing in one associative array. So I broke it up into chunks of 10.

Yeah, see we're still cruising there. So that just reads it all in. In fact, we can take a look at that file.

So I just took that exact same example and just put his data in-- so just to take a look at what that involved, pretty much all the same. It was one big file. But I couldn't process it.

I mean, I could read it in. He did a great job of creating it into triples. And I could easily hold those triples. But I couldn't quite construct the associative array.

And so I basically just go through and find all the separators, and then basically take them out of the vector in memory. And so that's what I'm doing here, is I'm looping over here. So, basically, I read in all the data. I find all the separators.

And then I go through. And it's a little bit of a messy calculation to basically do them in these particular blocks. And then I can construct the associative array and save those out, no problem.

And let's see here. So what else [INAUDIBLE]. We can do it in little degree calculation from that data. So it's now reading each one of those, and computing the degrees.

[INAUDIBLE] do the same thing on this system. It's pretty fast. Proceed then to-- let's create some tables. So I created a special class of tables for that. [INAUDIBLE] modify that [INAUDIBLE] that.

If you go over here, I think it was on this one. Nope, I did it on the other database-- database 1, got tables there. So all I was doing there was plotting the degree distribution.

So this shows us-- so, for each tweet-- let's go to figure one-- are you done? It's very Twitter-like. No, no one is ever done on Twitter. So-- wow, what did I do?

I went to town, didn't I? Done now? So we go to figure 1. You see for each tweet, this shows us how much information was in each tweet. And you see that, on average-- this is because he basically parsed out each word uniquely.

So this shows there is about 1,000 pieces of information associated with each tweet, which seems a little high. So we should probably double check that. And then what did I do?

So we loaded all of them up. We summed the degrees. And then-- oh, I said, show me all the locations with counts greater than 100, and then all the words with at signs greater than 100, and all the hashtags greater than 50. So that's what these other guys are.

So this was the-- essentially, for each tweet, how many do you have? If we go to figure 2, this shows the degree distribution of all the words and other things in there. So there's some guy here who is really, really high. In fact, we can find him out. But as, of course, most things occur only once-- like, there's a lot of unique keys. There's the message ID, which of course is probably something that only appears once.

If we go to figure 3-- so this just shows the locations. And this was tweets from the day before the hurricane, or the Wednesday before Hurricane Sandy. And so this shows us-- but limited to the New York area or something like that?

AUDIENCE: Yeah, 40 miles around New York City.

JEREMY KEPNER: 40 miles around New York City. Basically, this shows all the locations here. So this is a classic example of the kind of things you want to do, because the first thing that we see is that we

have some problems with our data, which is New York City and New York City space got to go in and correct all those. So that's a classic example.

This is really what D4M-- it's the number one thing that people do with D4M, is it's the first time that you really can do sums and tallies over the entire data. And these things just don't pop out. They stick out like a sore thumb.

Like, oh got to correct that. You can either correct it in the database, or correct it afterwards in the query. But that just immediately improves the quality of everything else you have.

And then there's this clutter one, like location none. Well, clearly, you'd want to just get rid of that, or do something with that, and then just plain old normal spelled New York. So most people can spell correctly. And so we're very good.

But location, iPhone, what's that about? Jersey City-- well, we don't care about that. South Jersey-- well, what's-- South Jersey people don't know that they're 40 miles from New York, I guess?

AUDIENCE: It's whatever they have.

JEREMY KEPNER: In their profile. So a lot of people in South Jersey who say they're from New York. So what's her name from Jersey Shore? Snooki, right Snooki says she's actually from New York City, not South Jersey.

So there's a great example of that. And then let's see here, figure 4. So this just shows all the at signs. So you see, basically, damnitstrue, is like the most-- is this like a retweeted thing or something?

I don't know. What does the at sign mean again?

AUDIENCE: It's to another user.

JEREMY KEPNER: To a user. So most people tweet to damnitstrue in New York. There you go. Funny fact, relatable quote, and then Donald Trump, the real Donald Trump, and then just word at sign. So those are another example-- another here is a hilarious idiot, badgalv, an Marilyn Monroe ID.

So there you go, a lot of fun stuff there on Twitter. But this is sort of-- he's going to establish his background, and then go back and look at during the hurricane. So this is all basically the

normal situation, very clearly.

And then the hashtags-- so we can look at the hashtags. So what do we got here? Favorite movie quotes.

AUDIENCE: Favorite movie quotes misspelled.

JEREMY KEPNER: And favorite movie quotes misspelled right up there. The Knicks, and then what I love the most, and all this type-- team follow back. I don't know, team auto-- no, what's this one? What's TFB?

Maybe we don't want to know. You can always look it up in Urban Dictionary. It's a bad one? All right, OK, good, we'll will leave it at that, won't add that to the video.

So continuing on here, let's see. Well, you get the idea. And so all these examples, they work in parallel to, you get a lot of speed up, lots of interesting stuff like that. But that's very classic the kind of thing you do.

You get data. You parse it. You maybe stick it in Matlab files to do your initial sweep through it. But then if it gets really, really big and you want to do more detailed things that you insert in the database, can do queries there.

Leverage using your counts, so that you don't accidentally-- you can imagine if we put all the tweets in the world and you had location, New York City. And you looked at-- you had to, give me all this set of locations. And one of them was New York City.

You'd be like, oh my God, I've just done a query that's going to give me 5% of all the data back. That's going to just flush your system. But if you can just do the count, and be like, oh, New York City has got a million entries. Don't touch that one. Or put an iterator on that one so that I only handle it in manageable chunks.

So I want to thank you. So hopefully this was worth it. We have one more class, which deals with a little bit of wrapping up some of the theory on this stuff, and some stuff on performance metrics.

And then in two weeks, for those of you who signed up, we have the Accumulo folks coming in showing you how to run your own database all day on just-- we're setting up a database for you guys on LLGrid. But you're definitely going to run in with your customers Accumulo

instances. It's good to know some basics about that, because a lot of times you're not going to have all the nice stuff that we've provided.

And it's good to know how to set up your own Accumulo and interact with that in the field. So with that, that brings the lecture to the end. And happy to stay for any questions, if anybody has them.