PROFESSOR: And so I want to thank you all for coming. As the holiday season comes along here I'm guessing that our people are getting distracted with other things. So a little bit more of an intimate setting today. So I urge people to take advantage of that, ask questions. It's often a lot easier to do when there aren't as many people in the room. So I just really want to encourage you to do that.

So let's get right into this. All right. Fortunately the mic actually doesn't pick that up at all for the most part. But if you all can hear the drilling of a-- yeah.

All right, so this is Lecture 4. I kinda went out of order to get our special Halloween lecture in last time. But we're going to talking a little bit more about the analysis of what we call structured data. And we're doing more structured type analyzes on unstructured data. And of course the signal processing on databases, this is where we bring the ideas of detection theory and apply them to the kinds of data we see in databases, strings, and other types of things there.

And so this lecture is going to talk a little bit more about-- we're going to get into a little bit more sophisticated analytics. I think up to this point we've done a fairly simple thing, basic queries using the technology, pretty simple correlations, pretty simple stuff. Now we're going to begin to kind of go a little bit more sophisticated in some of the things we do. And again, I think as things get more complicated the deform technology becomes, its benefits become even more apparent.

So we're just going to get right into this, gonna show you our very generic scheme, gonna talk a little bit more about some of the issues that we encounter when we deal with particular databases. But the database we're going to be using for this course is called accumulo. And so I'll be talking about some issues that are more accumulo specific.

So accumulo is a triple store. And so the way we read data into our triple store is using what we call this exploded transpose pair schema. So we have-- [INAUDIBLE]. We have an input

data set that might look like a table like this with maybe our row key is going to be somehow based on time, and then we have various columns here, which may or may not contain various data. And then the first thing we do is we basically explode, because our triple stores can hold an arbitrary number of columns. It can add columns dynamically without any costs. We explode this schema by appending our Column 1 and its value together in this way. We then just took a value, could be anything, like one, anything that we wouldn't want a search on.

And then we have a row key here. I'll get into back a little bit why we've done the row key in this way. And so by itself this doesn't give us any real advantage. But in accumulo or any other triple store end or in just in D form sociative arrays, we can store the transpose of the table, which means that all these columns are now rows. And this particular database is a row oriented database, which means it can look up any row key with constant access. So it can look up row keys very quickly. And D4M a lot we hide this from you so that whenever you do inserts it does, if you want it to happen, it will store the transpose for you. And so you look like you have a giant table where you can look up any row key or any column very, very quickly and makes it very easy to do very complicated analytics.

One little twist here that you may have noticed is that I have flipped this time field here, taking it from essentially little endian and made it big endian. And I've always been an advocate of having row keys that have some meaning in them. Sometimes in databases they just create your arbitrary random hashes. I think we have enough random data in the world that we don't necessarily need to create more, and so if we can have meaning in our row keys, I think it's very useful, because it just makes it that much easier to debug data that actually has meaning.

And so I've often advocated for creating having a row key be a timelike key. I think it's good to have timelike keys. And by itself there's nothing wrong with having a little endian row key, except when you go parallel.

If this database is actually running on a parallel system, you can run into some issues. People have run into these issues, which is why I now advocate essentially doing something else with the row key more like this. In particular, accumulo, like many databases, when it goes parallel it takes the tables and it splits them up. And it splits them up by row keys. So it creates continuous blocks of row keys on different processors.

If you have a little endian time row key, it means every single insert will go to the same processor. And that will create a bottleneck. And then what will happen over time it will then

migrate that data to the other processor. So it can be a real bottleneck if you have essentially a little endian row key.

If you have a big endian row key, if it just uses, it will then break up these things. And then when you insert your data, it will sort of naturally just cause that to spread out over all the systems. So that is if your data is coming in in some kind of time order, which happens. We definitely see people coming in that yesterday, you know, today's data comes in today and tomorrow's data comes in tomorrow. You don't want to have it all that data just hitting one processor or one compute node in your parallel database.

Other databases have more sophisticated distribution things they can use as sort of a round robin or a modular type of-- they'll create a hash that does a modular so that it doesn't have that, eliminates that type of hotspot. But for now we just recommend that people use this.

This does make it difficult to use the row key as your actual time value. And so what you would want to do is also stick, have a column called time that had the actual time in it. And then you could actually directly look up a time in that way. So that's just a little nuance there, good sort of design feature. Probably something that you wouldn't run into for quite a while, but we've definitely had people, hey I went off and implemented exactly the way you suggested, but now I'm going paralyzed seeing this bottleneck. And I'm like oh. So I'm going to try and correct that right now and tell people that this is the right way to do it.

So starting out simple, we're just going to talk about kind of one of the simplest analytics you can do, which is getting basic statistical information from your data set. So if this is our data set, now we have a bunch of row keys here, timelike with some additional stuff to make them unique, unique rows. And then we have these various columns here, some of which-- the gray ones are, I'm saying, are filled in, and the white ones are empty. And so what I want to do is just grab a chunk of rows.

OK, and then I'm going to compute basically how many times each column appears, essentially some. I'll sum by type. So I'll show you how we can look at if you want to compute just how many entries there were in column 1 or column 2, computing the covariance, computing type and para covariances are all different things that we can do.

So I'm going to do all those statistics on the next slide. All those analytics, which if you were to do them in another environment would actually be, you would write a fair amount of code. But here we can do them and each one of them is essentially a one liner.

So here's my set of row keys. OK? I've just created a list of row keys that have essentially a comma as the separator here. And there, service that.

And this is our Table T. So in the notation we often just refer Table T. This is in a binding to an Accumulator Table or it could be any table really, any database that we support. And so this little bit of code here says, return me all the rows given by this list of rows here and all the columns. So we're using the Matlab notation. Colon means return all the columns.

This will then return these results in the form of an associative array. Now since the values of that are strings, in this case maybe strings values of 1, we have this little function here called double logi, which will just basically say ignore whatever the value is and just make it-- if it's got an entry, give it a 1 or otherwise ignore it. So this is a shorthand. It basically does a logical, and then it does a double, so we can do math on it.

So this is our query that gets us our rows and returns them as an associative array with numeric values. We then compute the column counts. So that's just the Matlab sum command, which says basically this tells you the dimension that's being compressed. So it's compressing the first dimension with the rows. So it's basically collapsing it into a row vector. So it's just summing. So I just would tell us now we could then for all those rows count how many occurrences of each unique column, of each column type there was.

And then we can then get the covariance, the type type covariance by just doing A transpose A or square in. These do the same thing. This is usually slightly faster but not a lot. And so that just does the column type, column type, covariance, very simple.

And then finally we have this function, which essentially undoes our exploded schema. So let's say we wanted to return it back to the original dense format where we have essentially four columns, column 1, 2, 3, and 4. And we wanted to value put back into the value position. So we have this function call to type with this-- and I don't know if the name makes sense or not, but that's what we call it-- and basically just says, oh, this is the limiter. So it takes each one of those, splits it back out, and stuffs it back in. And so then you now have that associative array.

You can then do a sum again on that to just get like I want to know just how many column 1 instances, column 2 instance, column 3s there are. And likewise, just doing A transpose A or square in would then do the covariance of that. And that would tell you of those higher level types how many of there were.

So this is a lot of high level information. Really highly recommend that people do this when they first get their data, because it really uncovers a lot of bad data right from the get go. You'll discover columns that are just like, wow these two column types never appear together. And that doesn't make sense. There's something wrong there. Or why is this exploded column, why is it so common when that doesn't make sense? So again, it's very, very useful for doing things. And we always recommend people start with this analytic.

So again, that's very simple stuff. We've talked about it before. Let's talk about some more sophisticated types of analytics that we can do here.

So I'm going to build what I call a data graph. So this is just a graph in the data. It may not actually be a real. You may have some other graph in mind, but this is the data supports as a particular kind of graph in it.

So what we're going to do here is we're going to set a couple starting column here. OK, C0, that'll be our set of starting columns. And we're going to set a column. I said I've allowed column types. So we're going to be interested in certain column types. And we're going to create a set of clutter columns. These are columns that we want to ignore. We want to-- they're either very, very large or whatever.

And so the basic algorithm is that we're going to get all the columns. Our result is going to be called C1 here. OK, and that's going to be all rows in C0 that are of type CT, and excluding columns CL.

So this is a rather complicated joined type of thing that people often want to do. They want to do these types of things. Look, I want to get all these rows, but I only care about these particular types of columns. And I don't really care about, or I want to expressively eliminate certain clutter columns that I know are just like always pointing everywhere.

So let's go look through the D4M Code that does this sort of complicated type of join. So I'm going to specify my C0 here. And this could be a whole list. It's just the whole list of starting columns.

I'm going to specify my column types. In this case I'm going to have two column types. I'm gonna say, create a string called starts with, which essentially creates a column range, one column range around column 1, and one column range around column 3. And then I'm going to specify a clutter column here, which is just this A. And again, it could be a whole list as well.

All right, so step one here is I'm going to pass. I'm assuming that this table is bound to one of these exploded transposed pairs. So it will know when I give it columns to look up to then point to the correct table. So we have a C0 here. And it will say, all right, please return all rows, all the data that contains C0, basically all those rows.

I'm then going to say now I don't care about their values, I just want the rows. So this is an associative array, and this command row just says, just give me those rows. So basically I had a column, I returned all the rows, but now I just want the row keys.

I'm now going to take those row keys and pass them back into the row position. So now I will have gotten the entire row. So basically I got a column, I took those rows, and now I'm going and striping and painting and getting the whole rows.

So now I have every single. And since I don't care about the actual values, I just want them to be numeric, I just use the double logi command. So basically I've done a rather complicated little piece of query here in terms of get me all columns that contain, all rows that contain a certain column. And so that's now an associative array.

I'm then going to reduce to specific allowed columns. So I'm going to pass in. I'm going to say, please give me just the ones of these particular column types. So I got the whole row, but now I just want to eliminate those.

I could have probably actually put this in here, but whether it's more efficient or not to do it here or here, it's 6 of one, half dozen of the other. I try and make my table queries just either only columns or only rows. It tends to make things simpler. So now we do that. We have just those types.

And now I want to eliminate the clutter columns. So I have A, which is just of these type. And I want to then eliminate any of the columns that are in this, basically, column 1. So I had column 1 as one of my types, but I don't care about that. And so I can just do subtraction. I can just basically say, oh go get the clutter columns and subtract them from the data that I have.

And now I'm just going to get those columns, and I have now the set of columns. I now have C1. So I've completed the analytic I just described in the-- if you're grabbing like four lines rather sophisticating analytic, I could then proceed to look for additional clutter. For instance, I could then basically query say, please give me those, stick the C1 back in and then sum it up

and look for things that had a lot of values and continue this process. Just an example of something I might want to do when I have this data set. So these show you the kinds of more sophisticated analytics that you can do.

I want to talk a little bit about data graphs in terms of what are the things you can do here, what is supported. The topology of your data, remember this edge list has direction. When you have graphs they can very well have edges that have direction. And a lot of times people will say, well I want to get the whole graph. I'm like, well, if you're doing essentially things that are based on breadth first search, you may not be able to get the full graph, because you're never going at it from the correct direction. So you are definitely limited by the natural topology of your data.

So for example here, let's say I start with C0 as column 1A. So I now have essentially this vertex A let's call it. And now here I say, OK, I now want to get all. Give me the whole row of all I'm going to do. Give me all the rows that contain that.

All right, so then these two guys pop up. So I get another A and I get a B. So I got a B here. That's good.

And then I'm going to go, proceed, go down again. I'm like, all right. I'm going to then now say, give me all the rows that contain those columns. And I go down again, and I got the C. Did I get a C? No, I never got a C. I never got a C in any one of these, even though it's all in the data and probably all connected. I never actually got a C.

There I got the C. Now when I did it the second time I got the C, so there we go. So this is an example of a series of breadth first searches that result in getting the whole graph. But the graph had this typology and wouldn't naturally admit that. So certainly in this particular case the data and the queries were good for this, we would call a star, because essentially it's a vertex with everything going into it.

Let's take a different graph. This is what we call a cycle. So we see we have a little cycle here going like this. We start again with A. We get our columns. We get C1s across here. And that's kind of the end of the game.

We get A's, we get a B, but we're not going to get anything else when we column back up. We're not going to get anything else. And so we're basically stuck here at B. We weren't able to get to C or D.

So these are the kind of a little more subtle things that everyone has to worry about. And once you see it it's kind of, well, of course. I can't do-- I'm not going to get the whole graph. But you'd be amazed how many teams were like, I wanted the whole graph, and I just can't do it. It's like, well, you don't have the edges going in the right direction for you to do that. You're going to have to think things through a little bit more. So I just want to-- it's kind of a little catch that I want to point out to people, because it's something that people can run into.

We're going to do a little different type of analytic here. I've changed some of my columns here. I have some. Let's call these coordinates. I'm going to have now with my data set an x and a y-coordinate that I'm storing in different rows and columns. I want to do some kind of space windowing. I want to find all data within a particular x and y-coordinate.

So what I'm going to do is I'm going to select a set of data here R instead of rows. And I'm going to give a space polygon. And I'm going to query, get the data. And then I'm going to extract the space coordinates from the values there, and I'm gonna return all columns that are within my space window here. And again, this is good for finding columns in between your space window.

If you're concerned that you're going to be getting an awful lot of if you have, let's say you have a coordinate that goes through New York. And you're concerned that's just going to-- you don't want to get, you don't want New York, but you happen to be on the same latitude and longitude of New York. You can do something called Mertonization, which basically is essentially imagine taking your strings of your coordinates and interleaving them. And now you've essentially created an Ascii based grid of the entire planet. And so that's a way of if you want to quickly filter down, you can get a box and then go back and do the detailed coordinates to vent yourself from having to do.

So that's a standard trick. And there's a variety of Mertonization schemes that people use for interleaving coordinates in this way. I think Google Earth has a standard box now as well.

I find this the simplest, because you literally just take the two strings and interleave them. And if you have a space, you can actually then do with variable precision, because if you just like, space I don't know. And it all kind of works out pretty nicely.

And you can read the coordinate right there. Like, the first one is the first. And you can even include the plus and minus signs for a lot if you wanted to. So maybe not the most efficient

scheme, but one that's human readable. And I'm a big advocate of human readable types of data schemes.

All right, so let's actually do that analytic. So again, I created my-- selected my row, got my x and y-coordinates in those rows, and then figured out which columns they were that satisfied that. So let's do that now in code. Let's see here.

So we have the-- all right, so we have, in this case I gave It a row range. So you can do-- this is what a range query looks like. If you give any, either an associative array or a table, something that is a triple, essentially that's a string, colon, and another string, it will treat that as a range query. And we actually support doing if you have multiple sets of triples, it should handle that, which is good.

I'm going to specify my bounding box here, essentially a box. And I happen to do with complex numbers just for fun, just because complex numbers are a nice way to store coordinates on a two dimensional plane. And Matlab supports them very nicely. Complex numbers are our friends, so there you go. But I could have just as easily had a set of x and y vectors.

So I'm going to get all the rows. So I query that. Very good, we have that. And then, so that just gives me that set of data, all those rows. I then use my starts with my x and y to get just the columns of those X's and Y's. And I'm now kind of going to convert those exploded back into a regular table with this call to type function. So that basically takes those values. So it takes those coordinate values, so like we saw. It takes these coordinate values here like the 0 1, and it puts it back into the value position.

So now I have, though, it will still be a string in the value position, which our associate arrays can handle fine. But I now want to really treat it like a number. So we just have overloaded the standard Matlab string to numb function, which will convert those strings and will store them back. You now have an associated array with numbers in them.

So we call this Axy. And now we can do something. We basically can extract the x values here. So we have Axy, and say, all right, give me the x column, and then Axy and give me the y column. And Matlab has a built in function called in polygon, which you give it a polygon. So I give it the real and the imaginary parts of my polygon here S and the x and y-coordinates. And it will return in essentially the value of is that in there, which is great, because there are many dissertations written on the in polygon problem. And it's nice that we have a nice built in Matlab function to do that.

And then now I have that, and I can just pass that back into the original A. So I do find. This actually returns a logical of zeros and ones. If I do find, then that will return a set of indices. And I just pass those indices back into A. And then I can get those columns and there we go, all very standard Matlab like syntax.

Again, this is a fairly complicated analytic. If you were doing this using other technologies, I mean, I'm sure all of us would be writing a fair amount of code. And this is just the kind of thing that we can do very easily in D4M.

Another analytic, which is probably a bit of a stretch, but I was just having fun here, is doing convolution on strings, which is a little odd. But I gave it I gave it a whirl.

So what we want to do is we want to convolve some of our data with a filter. I mean, convolving the filters as a standard type of thing. So it's the standard way we do detection here. And so the way we do that is once again, I give a list of rows that I want here. I'm going to create a filter, which is just essentially a box of 4 wide box.

So I get my rows. And then I convert them to numeric. And I'm going to do my convolution on the x columns. So let's see here. So I'm going to get these. I'm basically getting all the x-coordinates. I'm going to sum all of those, so I basically now have all those. And now I'm going to pop those back into their values. And now I can do a convolution.

And this convolution works if one of the axes is sort of like an integer sequence type of thing. So you can do-- it tries to extend that naturally. So something to play around with if you want to do convolutions. We sort of support it. And I'm sure if anyone of you do play around it, we would be glad to hear your experiences, think about how we should extend it.

So these are all sort of basic standard first order statistical analytics that one can do on data sets. And we can support them very, very well. Let's do some more complicated what I would call second order analytics.

So I'm going to do something called, it's a complicated join essentially, what I call a type pair. So what I want to do here is I want to find all rows that contain values. I want to find rows that have both value of type 1 and of type 2. So I'm going to specify this to be, basically x to be type 1 and y to be type 2. And I want to find all data that has entries in both those very, very standard type of join type of thing.

And this is done a little bit more complicated than we need it to be just to show you kind of some of the richness here. You can kind of take a fork in any way. We could probably do this whole thing in about two lines, but I'm kind of showing you some additional features of D4M in the spirit of this analytic.

So again, I'm just going to use a range query here. So I have this range. I'm going to have my type 1 be starts with x, and my type 2 be starts with y. So I do my query. I convert all the string 1's to numeric 1's. And then what I'm going to do is I'm going to basically, all right, get me all the columns of type 1, sum them all together, find everything that equals-- and I only care about the 1's that are exactly equal 1. So like if I had two x's, I'm like no. I don't want those. I want exactly one x in this row.

And then I'm going to take those rows that have exactly one x. I'm going to pass them back into A. So I now just get the rows that have exactly one x. I'm going to filter it again with ct1 and ct2, although, I don't need to do that. Then I'm going to sum it again.

And now I'm going to say, all right, give me the only ones that are exactly 2. So I know my x is exactly 1. So in order for it to be exactly 2, that means my y also had to have only exactly one entry in it. So now I have the data that just has exactly one of each of those.

Now I want to create sort of like a cross-correlation pair mapping of this. So I'm actually to look for x's across columns that appear say, I want to look for x's that appear with more than one y or a y that appears with more than one x.

So there's a variety of ways to do that. Here what I'm doing is-- so I have gotten the rows of A that have exactly one x and y. I now pass that back again to get my C, to get the x's again. And one of the things that we've done that's kind of nice is we've overloaded the syntax of this query on our associated arrays and on our table queries such that if it only has one argument, it will return an associative array. But if you give it three output arguments, it will return the triple, so in this case, the row, the column, and the value.

Now I don't care about the row and the value, I just care about the column. But that's a nice way. We're often gonna in certain cases want to bump back and forth between the triples representation and the associated array implementation. Now you always can use the Find command around any associated array just as you can in normal Matlab matrices not return the triple.

The advantage of doing it here is that it's faster, because what we actually do when we do the query internally is we actually get the triples and then convert to an associative array. And if you just say I want the triples, we can just short cut that and give you the triples right away. So sometimes if you're dealing with a very large associative arrays or some operation that's just I want to get some more performance back then working. Especially if you like, well, I only care about one thing, I don't care about all of the values, I don't really need to be a full associative array, then that's a great way to sort of short circuit that.

So we do that here. And now we can construct a new associate array, which is just taking the x's and the y's and creating a new associative array with those. And that just shows me the correlations between the x's and the y's. And I can then find ct, basically x's that have more than one y-- so I've just summed them there-- or y's with more than one x's.

Again, these are very similar to analytics that people want to do. And again, very simple to do. And again, showing you some of the different types of syntax that are available to you in D4M. Again, if you're used to using Matlab, these types of tricks are very natural. We're just showing you that they also exist within D4M.

So here's another one. So I wanted to find column of pair set C1 and C2, and get all columns C1 and C2, and find the rows that have just one entry in C1 and C2. And it basically checks to see if data pairs are present in the same row. Again, something that people often want to do. You've got a complicated type of join.

So here we have a set of columns one C1, a set of column C2. I want to create this, I want to sort of interleave these together into a pair. So I want to create some concept of a pair. And so we have this function here called cat string, which basically will take two strings and another delimiter, and will basically if they are of the same number of strings will just leave them together. If one of these is just a single string, it will just an essentially preappend or append that.

So for instance, if you are wondering how we actually create these exploded values like call 1 and B, that's just basically using this function here. We get the values, we get the columns, we put essentially the pipe thing in the middle, and it just merges them together. So we now sort of interleave these two together. So we'll now have something like Col1 b pipe Col3 b pipe comma as the separator.

So now I can create a set of pair mappings from C1 to its pairs. OK, that's A1 to its pairs and

A2 to it's pairs. I can get the columns of those A1 and A2. And then I can find all the pairs by essentially through this combination of matrix multiplies and additions and etc. So a very sort of complicated analytic done very nicely.

And then there's a whole bunch of different ones you can do here. These are almost semantic extension. The column types may have several different types, and you want to do. So for instance, if I have a pair of columns here, column 1 and column 3, I could say, well, that also implies this. Column three equals column 1. That's one kind of sort of pair reversal type of thing.

You'll have extensions. You might say, look if I have a column 1A that also implies that really there should also be a column 2A, and other types of things like that. So these are just types of things that people do with pairs. They're often very useful. And I think that basically brings us to the end of the lecture portion of class.

So again, just the exploited schema really allows you to do this very rapidly with your data. And you can implement a very efficient graph analytics as a sequence of essentially row and column queries, because we use this very special exploded transpose pair schema. And increasingly as you become more and more skilled with this, you will discover that many, many, many of your analytics really reduce to matrix matrix multiplies. That matrix matrix multiply really captures the whole sort of all correlation that you want to do without having to kind of figure things out.

All right, so I'm now going to go and show some not these specific analytics, but some analytics that are more sophisticated based on the Reuters data set. If you remember a few weeks ago, we worked with the Reuters data set. And so let's see here. So we already did the entity analysis application a few weeks ago. I'm going to now do basically what happens when you construct tracks, which is a more sophisticated structured analytic.

And the assignment I'll send out was basically doing more cross correlations. For those of you who have kept it going here this far and continue to do the homeworks, I'll send this homework out to you, which is basically just cross correlating the data sets that you have. Again, and not an assignment that really predispose requires you having done the previous assignments. Just any data set, pull it into an associate array, and then do matrix multiplies to figure out the cross correlations and what they mean.

All right, so with that, why don't we take a short five minute break. And then I'll come back and

show you the demo.