JEREMY KEPNER: Thank you all for coming. I'm glad to see that our attendance after the first lecture has not dropped dramatically-- a little bit. That's to be expected. But so far I haven't managed to scare you all off.

So this is the second lecture of the signal processing on database of course. The format of the lecture will be the same as the last one. We'll have some slide material that will cover the first half of the course. And then we'll take a short break. And then there'll be some example demo programs.

And so with that, we'll get into it. And all this material is available in your LLGrid accounts. In the Tools directory you get a directory like this. And the documents and the slides are right here. So we'll go to lecture 01.

For those of you who weren't there or to recap for those who will be viewing this on the web later, the title of the course is Signal Processing on Databases-- two terms that we often don't see together. The signal processing element is really alluding to detection theory, and the deeper linear algebra basis of detection theory. The databases is really alluding to unstructured data, data that we often represent with strings and words and sort of sparse relationships. And so this course is really bringing these two normally quite separate ideas and bringing them together in a way because there's a lot of need to do so.

So again, this is the first course ever taught on this topic. There really isn't a prior model for how to teach this. So the approach that we're taking is we're going to have a lot of examples that show how people use this technology and hopefully see that. And then we'll also have some theory as well as a lot of examples. And so hopefully through some combination of those different types of teaching vehicles, each of you will be able to find something that is helpful.

So the outline of today's lecture is I'm going to go through an example where we analyzed some citation data. So citation data is one of the most common examples used in the field because it's so readily available and so easy to get. And so there's no issues associated with

it.

So you have lots of academics analyzing themselves, which is, of course, very symmetric, right? And so there's a lot of work on citation data. We are going to be looking at some citation data here, but doing it using the technology, the D4M technology, that you all have access to, doing it in the way that would work well there, which we think is more efficient both in terms of the performance that you'll get on your analytics and just in terms of the total time it will take you. Of course, it does require a certain level of comfort with linear algebra. That's sort of the mathematical foundation of the class.

So we're going to get right into it. And I talked about this in the previous lecture, these special schemas that we set up for processing this kind of data that allow us to, essentially, take any kind of data and sort of pull it into one basic architecture that is a really a good starting point for analysis. So here's a very classic type of data that you might get in citation data here.

So in this particular data set, we have a universal identifier for each citation. So every single citation is just given some index in this data set. And we'll have an author, a doc ID, and then a reference doc ID. So this basically is the document ID of the document that's being referenced and the one that the reference is coming from. That may seem a little opposite. But that's sort of what it means.

And you can have absences of both. You can have references where it's referring to another document for which they have not constructed some kind of document ID. You can also have references that are within a document that just for some reason or other doesn't have a document ID as well. So I think the important thing to understand here is that this kind of incompleteness on a data set, which might be as clean a data set as you would expect, which is the scientific citation data set, this kind of incompleteness is very normal.

You never get data sets that are even remotely perfect. To get good data sets that feel even like about 80% solid, you're doing very well. And it's very, very common to have data sets that are incomplete, that have mistakes, null values, mislabeled fields, et cetera, et cetera. It is just a natural part of the process.

I think even though more and more of these sets are created by machines and you're hoping that would eliminate some of those issues, well, the machines have the same problems. And they can make mistakes on a much larger scale, more quickly than humans can. So incompleteness definitely exists.

We pull this data into-- this data is sort of a standard SQL or tabular format here, in which you have records and columns and then values. And what we tend generally do is pull it into what we call this exploded schema where we still use the records. Those are the row keys. We create column keys, which are essentially the column label and the column value appended together. So every single unique value will have its own column.

And then we sort of build it out that way. And this creates this very, very large, sparse structure. Once we store the transpose of that in our database as well, we get, essentially, constant-time access to any string in the database. So we can now look up any author very quickly, any document very quickly, any record very quickly. And so it's a very natural schema for handling this kind of data.

Typically, we just hold the value 1 in the actual value of the database. We can put other numbers in there. But it generally should be a number that you would never want to search on because in many of the data sets that we have or the databases we're working with, they can look up a row key or with this schema, a column key very quickly. But to look up a value requires a complete traversal of the table, which is, by definition, very expensive. So there's definitely information you can put in there but just general information you would never want to really look up.

In addition for this particular data set, we have some fields that are quite long. And so we might have the fully formatted reference, which in a journal article can be quite long. You know if it's got the full name and the journal volume and maybe all the authors and all that stuff, that can be quite long.

It's very unstructured. We might have the title. We might have the abstract. These are longer blobs of data that maybe our other data set essentially is extracted all the key information from that. But often, we still want to have access to this information as a complete uninterrupted piece of data.

So if you actually find a record, you're like, well, I'd like to see the abstract as it was actually formatted as it was in the journal. Because a lot of times when you pull this stuff out and do keywords, you might lose punctuation, capitalization, other types of things that just make it generally easier to read, but don't really help you with indexing. So we actually recommend that that kind of information be stored in a completely separate table-- a traditional table, which would just be some kind of row key and then the actual reference with the values there.

You're never going to search the contents of this. But this allows you to at least have access to the data. So this is sort of a small addition to our standard schema where we have the exploded schema with all the information that we would ever want to search on with this transpose, which gives us a very fast search.

You may have an additional table which just stores the data in a raw format separately. You can look it up quickly. But it won't get in the way of any of your searches. The problem can be if you're doing searches and you have very large fields mixed in with very small fields, it can become a performance bottleneck to be handling these at the exact same time.

Another table that we used in this data set, which was to essentially index all the words in the title and the abstract and their Ngrams, and so the one gram would just be the individual words in a title. If a word appeared in a title, you would create a new column for that word with respect to that reference. An Ngram would be word pairs. A trigram would be all words that occur together in threes and on and on and on. And so often for text analytics, you want to store all of these.

And essentially, you might have here an input data set with various identifiers-- a title that consists of a set of words, an abstract that consists of set of words. And then typically what we would do is we would say for the one gram of the title a, we might format it like this. And then here we might hold where that appeared in the document. So this allows you to, essentially, come back and reconstruct the original documents.

So that's a perfectly good example of something that we don't tend to need to search on, which is the exact position of a word in a document. We don't tend to really care that the word was in the eighth position or the 10th position or whatever. So we can just create a list of all of these values.

And that way, if you want to then later reconstruct the original text, you can do that. Between a column and its positions, you have enough information to do that. So that's an example of another type of schema that we admit and how we handle this type of data for doing these types of searches.

And now it makes it very easy to say, show me, if I wanted to look up, please give me all documents that had the word d, this word d, in the abstract, I would simply go over here, look up the row key, get these two columns, and then I could take those two column labels and

then look them up here. And I would get all the Ngrams. Or I could then take those row keys and look them up in any of the previous tables and get that information. So that's kind of what you can do there.

I'm going over this kind of quickly. The examples go through this in very specific detail. But this is kind of just showing you how we tend to think about these data sets in a way that's very flexible. But showing you some of the nuances. It's not just the one table. It ends up being two or three tables. But still, that really covers what you're looking for.

Again, when you're processing data like this, you're always thinking about a pipeline. And D4M, the technologies you talk about is only one piece of that pipeline. We're not saying that the technologies we're talking about cover every single step of that pipeline.

So in this particular data set that we had, which I think was 150 gigabyte data set, step one was getting a hard disk drive sent to us and copying it. It came as a giant XML zip file. Obviously we had to uncompress it. So we had to zip file. We uncompressed it.

And then from there we had to write a parser of the XML that then spat that out into just a series of triples that then could be inserted into our database the way we want. Well, D4M forum is very good for developing how you're going to want to parse those triples initially. If you're going to do a high volume parsing of data, I would really recommend using a low-level language like C. Generally C is an outstanding language for doing this very repetitive, monotonous parsing and has excellent support for things like XML and other types of things. And it will run as fast as you can.

Yeah, you end up writing more code to do that. But it tends to be kind of a do once type of thing. You actually do it more than once, usually, to get it right. But it's nice when, I think, our parser that we wrote in C++ could take this several hundred gigabytes of XML data and convert it in triples in under an hour. And so that's a nice thing.

And other environments, if you try and to do this in, say, Python or other types of things or even in D4M at high volume, it's going to take a lot longer. So the right tool for the job-- it will take you more lines of code to do it. But we have found that it's usually a worthwhile investment because it can take a long time.

Once we have the triples, typically what we then do is read them into D4M and construct dissociative arrays out of them. Talked a little bit about associative arrays last time. I'll get to

that again. But these are essentially a MATLAB data structure. And then we generally save them to files.

We always recommend right before you do an insert into a database, even when you write performance parsers or if you're not writing, the parsing can be a significant activity. And if you write the data out in some sort of efficient file format right before you insert it, if you ever need to go back and re insert the data, it's now a very, very fast thing to do. You'll often have databases especially during development, they can get corrupted. You try and recover them but they don't recover.

And again, just knowing that you have the parsed files laying around someplace and you can just process them whenever you want is a very useful technique and usually is not such a large expense from a data perspective and can allow you essentially reinsert the data at the full rate the database can handle, as opposed to having you redo your parsing, which sometimes can take 10 times as long. We've definitely seen many folks who do full parsing, text parsing in Java or in Python that often is 10 times slower than the actual insert part of the database. So it's nice to be able to checkpoint that data.

And then from there we read the files in and then insert them into our three two database pairs-- the keys and the transpose, the Ngrams and the transpose, and then the actual raw text itself. I should also say, keeping these associative arrays around as files is also very useful. Because the database, as we talked about before, is very good if you have a large volume of data and you want to look up a small piece of it quickly.

If you want to re-traverse the entire data set, do some kind of analysis on the entire data set, that's going to take about the same time as it took to insert the data. So you're going to be having to pull all the data out if you're going to be wanting to analyze all of it. There's no sort of magic there.

So the database is very useful for looking up small pieces. But if you want to traverse the whole thing, in which case having those files around is a very convenient thing. If you're like, I'm going to traverse all the data, I might as well just begin with the files and traverse them. And that's very easy.

The file system will read it in much faster than the database. The bandwidth of the file systems are much higher than databases. And it's also very easy to handle in a parallel way. You just send different processes, parse-- in reading in different files. So if you're going to be doing

something that traverses an entire set of data, we really recommend that you might do that in files.

And in fact, we even see folks its like, well, I might have a query that requires 3% or 4% of the data. Well then we would recommend, oftentimes, if you know you're going to be working on that same data set over and over again, querying it out of the database, saving to a set of files, and then just working with those files over and over again. Now you never have to deal with the latencies or other issues associated with the database. You've completely isolated yourself from that.

So databases are great. You need to use them for the right thing. But the file system is also great. And often in the kind of work that we do, that algorithm development people do, testing and developing on the files is often better than using the database as your fine-grain rudimentary access type of thing. So we covered that.

So this particular data set had 42 million records. And just to kind of show you how long it took using the pipeline, we uncompressed the XML file in one hour. So that's just running gzip. Our parser, that converted the XML binary structure into triples, this was written in C#, was two hours. So that's very, very nice.

As we debug the parser, we could rewrite it and not have to worry about, oh my goodness, this is going to be days and days. The constructing of the D4M associative arrays from the triples took about a day. That just kind of just shows you there.

And then inserting the triples into Accumulo, this was a single node database. Not a powerful computer, it was a dual core circa 2006 Server. But we have sustained an insert rate of about 10,000 to 100,000 entries per second, which is extremely good for this particular database, which was Accumulo.

So inserting the keys took about two days. Inserting the text took about one day. Inserting the Ngrams took about 10 days in this particular data set. We ended up not using the Ngrams very much in this particular data set, mostly the keys.

And so there you go. That gives you an idea. If the database itself is running in parallel, you can increase those insert rates significantly. But these are the basic single node insert performance that we tend to see.

Yes, question over here.

AUDIENCE: Silly question. Is Accumulo a type of a database?

JEREMY KEPNER: Ah yes. So I talked about that a little bit in the first class. So Accumulo is a type of database called a triple store or sometimes called a NoSQL database. It's a new class of database that's based on the architecture published in the Google Bigtable paper, which is about five or six years old.

There's a number of databases that have been built with this technology. It sits on top of a file system infrastructure called Hadoop. And it's a very, very high performance database for what it does. And that's the database that we use here. And it's open source, a part of the Apache project. You can download it.

And at the end of the class, we'll actually be doing examples working with databases. We are developing an infrastructure, part of LLGrid, to host these databases so that you don't really have to kind of mess around with the details of them. And our hope is to have that infrastructure ready by the last two classes so that you guys can actually try it out, so.

So the next thing we want to do here is after we've built these databases, I want to show you how we construct graphs from this exploded schema. Formally, this exploded schema is what you would call an incidence matrix. So in graph theory, graphs generally can be represented as two different types of matrix-- an adjacency matrix, which is a matrix where each row and column is a vertex. And then if there is an edge between two vertices, there'll be a value associated with that. Generally very sparse in the types of data sets that we work with.

That is great for covering certain types of graphs. It handles directed graphs very well. It does not handle graphs with multiple edges very well. It does not handle graphs with hyper-edges. These are edges that connect multiple vertices at the same time.

And the data that we have here is very much in that latter category. A citation, essentially, is a hyper-edge. Or basically one of these things it connects a document with sets of people, with sets of titles, with sets of all different types of things. So it's a very, very rich edge that's connecting a lot of vertices.

And so a traditional adjacency matrix doesn't really capture it. So that's why we store the data as an incidence matrix, which is essentially one row for each edge. And then the columns are vertices.

And then you basically have information that shows which vertices are connected by that edge. So that's how we store the data in a lossless fashion. And we have a mechanism for storing it and indexing it fairly efficiently.

That said, the types of analyses we want to do, tend to very quickly get us to adjacency matrices. For those of you who are familiar with radar-- and this is probably the only place where I could say that-- I almost think of one as kind of the voltage domain and the other the power domain. You do lose information when you go to the power domain.

But often, to make progress on your analytics, you have to do that. And so the adjacency matrix is a formal projection of the incidence matrix. You essentially, by multiplying the incidence matrix by itself or squaring it, can get the adjacency matrix.

So let's look at that. So here's an adjacency matrix of this data that shows the source document and the cited document. And so this just shows what we see in this data--

--cornor up here. If you see my mouse going up here, scream. So we'll stay in the safe area. So This is the source document. And this is the cited document.

This is a significant subset of this data set. I think it's only 1,000 records. It's sort of a random sample. If I showed you the real, full data set, obviously that it just be solid blue.

The source document ID is decreasing. And basically, newer documents are at the bottom, older documents at the top. I assume everyone can know why we have the sharp-edged boundary here.

AUDIENCE: Can you cite future documents?

JEREMY KEPNER: You can't cite future documents, right? This is the Einstein event cone of this data set, right? You can't cite documents into the future. So that's why we have that boundary there.

And you could almost see it here. You see-- and I don't think that's just an optical illusion-- you see how it's denser here and sparser here, which just shows as documents get published over time, they get cited less and less going into the future. I think one statistic is that if you eliminated self-citations, most published journal articles are never cited. So thank God for self-citation.

This shows the degree distribution of this data. We'll get into this in great much greater detail.

But this just shows here on this small subset of data, basically for each document we count how many times it was cited. So these are documents that are cited more. And then we count how many documents have that number of citations.

10 to the 0 is basically documents that are cited once. And there shows the number of documents in this set here, which was 20,000, 30,000, 40,000 that have one citation. And then going on down here to the one that is cited the most. This is what is called a power law distribution.

Basically it just says most things are not cited very often and some things are cited a long time. You see those are on a approximately linear negative slope in a log-log plot. This is something that we see all the time, I should say, in what we call sub-sampled data where essentially the space of citations has not been fully sampled. If people stopped publishing and we created new documents and all citations only were of older things, then over time you would probably see this thing begin to take on a bit more of a bell-shaped curve, OK? But as long as you're an expanding world, then you will get this type of power law.

And a great many of the data sets that we tend to work with fall in this category. And, in fact, if you work with data that is a byproduct of artificial human-induced phenomena, this is what you should expect. And if you don't see it, then you kind of want to question what's going on there. There's usually something going on there.

So those are some overall statistics. Those are two things that we-- the adjacency matrix and the redistribution of two things that we often look at. I'm going to move things forward here and talk about the different types of adjacency matrix we might construct from the incidence matrix.

So in general, the adjacency matrix is often denoted by the letter A. And incident matrices, you think we might use the letter I. But I has obviously been taken in matrix theory. So we tend to use the letter E for the edge matrix. So I used E as the associative array representation of my edge matrix here.

And if I want to find the author-author correlation matrix, I just basically can do E starts with author transpose times E starts with author. So basically the starts with author is the part of the incidence matrix that has just the authors. I now have an associative array of just authors. And then I square it with itself. And you get, obviously, a square symmetric matrix that is dense along that diagonal because every single document, the author appears with itself.

And then this just shows you which authors appear with which other author. So if two authors appear on the same article together, they would get a dot. And so there's a very classic type of graph, the co-author graph. It's well studied, has a variety of phenomena associated with it. And here, it's constructed very simply from this matrix-matrix multiply.

I'm going to call this the inner square. So we actually have a short cut for this in D4M called [INAUDIBLE]. So [INAUDIBLE] means the matrix transposed times itself, which sort of has an inner product feel to it. No one has their name associated with this product. Seems like someone missed an opportunity there.

All of the special matrices are named after Germans. And so some German missed an opportunity there back in the days. Maybe we can call it the Inner Kepner product. That would sufficiently expire it-- obscure it like the Hadamard product, which I always have to look up.

And here is the outer product. So this shows you the distribution of the documents that share common authors. So it's the same. What I'm trying to show you is these adjacency matrices are formally projections of the incidence matrix onto a subspace. And whether you do the inner squaring or the outer squaring product, those both have valuable information.

One of them showed us which authors have published together. The other one shows us which documents have common authors. Both very legitimate pieces to analyze. And in each case, we've lost some information by constructing it. But probably out of necessity, we have to do this to make progress on the analysis that we're doing.

And continuing, we can look at the institutions. So here's the inner institution squaring product. And so this shows you which institutions are on the same paper. And then likewise, this shows us which documents share the same institution. Same thing with keywords, which pairs of keywords occur in the same document, which document share the same keywords, et cetera, et cetera, et cetera.

Typically, when we do this matrix multiply, if we have the value be a 1, then actually that the value of the result will be the count. So this would show you how many keywords are shared by the document. Or in the previous one, how many times a pair of keywords appear in the same document together. Again, valuable information for constructing analytics. And so these inner and outer Kepner products or whatever you want to call them, are very useful in that regard.

So now we're going to take a little bit of a turn here. This is really going to lead you up into the examples. So we're going to get back in to this point I made and kind of revisit it again, which is the issues associated with adjacency matrix, hypergraphs, multi-edge graphs, and those types of things.

I think I've already talked about this. Just to remind people, here's a graph, a directed graph. We've numbered all the vertices. This is the adjacency matrix of that graph.

So basically two vertices have an edge. They get a dot. The fundamental operation of graph theory is what we call breadth-first search. So you start at a vertex and then go to its neighbors, following the edges.

The fundamental operation of linear algebra is vector matrix multiply. So if I construct a vector with a value in a particular vertex and I do the matrix multiply, I find the edges and I result in the neighbors. And so you have this formal duality between the fundamental operation of graph theory here, which is breadth-first search and a linear algebra. And so this is philosophically how we will think about that. And I think we've already shown a little bit of that when we talk about these projections from incidence matrices to adjacency matrices.

And again, to get into that a little bit further and bring that point home, the traditional graph theory that we are taught in graph theory courses tends to focus on what we call undirected, unweighted graphs. So these are graphs with vertices. And you just say whether there's a connection between vertices. And you may not have any information about if there's multiple connections between vertices. And you certainly can't do hyper-edges.

And this gives you this very black and white picture of the world here, which is not really what the world really looks like. And in fact, there's a painting courtesy of a friend of mine who is an artist who does this type of painting. I tend to like it. And this is the same painting of what I showed before, but what it really looks like.

And what you see is that this is really more representative of the true world. Just drawing some straight edge lines with colors on a canvas we already have gone well beyond what we could easily represent with undirected, unweighted graphs. First of all, our edges in this case have color. We have five distinct colors, which is information that would not be captured in the traditional unweighted, undirected graph representation.

We have 20 distinct vertices here. So I've labeled every intersection or ending of a line as a

vertex. So we have five colors and 20 distinct vertices.

We have 12 things that we really should consider to be multi-edges, OK? These are essentially multiple edges that connect the same vertices. And I've labeled them by their color here.

We have 19 hyper-edges. These are edges that fundamentally connect more than one vertex. So if we look at this example here, P3 connects this vertex, this vertex, this vertex, and this vertex with one edge. We could, of course, topologically say, well, this is the same as just having an edge from here to here or here to here to here. But that's not what's really going on. We are throwing away information if we decompose that.

Finally, we often have a concept of edge ordering. The order that the line is drawn here, you can infer it just from the layering of the lines that certain edges occur before others. And ordering is also very important. And anyone want to guess what color was the first color of this painting? When the artist started, they painted one color first. Anyone want to guess what that color was?

AUDIENCE: Red.

JEREMY KEPNER: What?

AUDIENCE: Brown?

JEREMY KEPNER: It was an orange. It was orange. And I would have never have guessed that except by talking to the artist, which I found interesting. And when she was telling me how she did it, it was so counterintuitive to the perception of the work, which I think is very, very interesting.

So if you were to represent this in a standard graph, you would create 53 standard edges. You would say this hyper-edge here, we would break up into three separate edges. This hyper-edge you would break up into two separate edges et cetera, et cetera. So you have 53 standard edges.

So one of the basic observations is that the standard edge representation, the fragments, or hyper-edges, a lot of information is lost. The digraph representation compresses the multi-edges. And a lot of information is lost. The matrix representation drops the edge labels. And a lot of information is lost.

And the standard graph representation drops the edge order. And again, more information loss. So we really need something better than the traditional way to do that.

And so the solution to this problem is to use the incidence matrix formulation where we assign every single edge a label, a color, an order that it was laid down. And then for every single vertex we can-- so this edge B1 is blue. It's in the second order group. And its connects these three vertices. And you see various structures here appearing from the different types of things.

So this is how we would represent this data as an incidence matrix in a way that preserves all the information. And so that's really kind of the power of the incidence matrix approach to this. All right. So that actually brings us to the end of the first-- oh, so we have a question here.

AUDIENCE: So this is great, and I've worked with [INAUDIBLE] before. But what kind of analysis can you actually do on that?

JEREMY KEPNER: So the analysis you typically do is you take projections into adjacency matrices. Yep. Yep. So this way, you're preserving the ability to do all those projections, you know? And it's not to say sometimes to make progress if your data sets are large, you're just forced to just make some projections to get started.

Theoretically, one could argue that the right way to proceed is you have the incidence matrix. You derive the analytics you want. Maybe you use adjacency matrices as an analytic intermediate step. But then once you've figured out the analysis, you then go back and make sure that the analytic runs just on the raw instance matrix itself and save yourself the time of actually constructing the adjacency matrix.

Sometimes this space is so large, though, it can be difficult for us to get our mind around it. And sometimes it can be useful just like you know what, I really think that these projections are going to be useful ones to get started. Rather than trying to just get lost in this forest, sometimes it's better to be like I'm just going to start by projecting.

I'm going to create a few of these adjacency matrices to get started, make progress, do my analytics, and then figure out if I need to fix it from there. Because working in this space directly can be a little bit tricky. Every single analysis-- it's like working in the voltage domain. You're constantly having to keep extra theoretical machinery around. And sometimes to make progress, you're better off just going to the power domain, making progress, and maybe

discover later if you can fix it up doing other types of things.

Are there other questions before we come to the end of this part of the lecture? Great. OK, thank you. So we'll take a short break here. And then we'll proceed to the demo.

There is a sign-up sheet here. If you haven't signed up, please write it down. Apparently this is extremely important for the accounting purposes of the laboratory. So I would encourage you to do that. So we'll start up again in about five minutes.