JEREMY KEPNER: All right. I want to thank you all for coming to the third lecture of the Signal Processing on Databases class. So we will go-- just to remind folks, all the material is available in the distribution of the software, which you all have in your [INAUDIBLE] accounts. Here, we're going to go to lecture two.

Courses. Course, which brings it together. Signal processing, which is based on detection theory, which is based on linear algebra with databases, which are based on strings and searching and really bringing those two concepts together in the course. So that we can use the mathematics we know from detection theory and apply it into new domains.

And I'm glad to see we have a full house again. So far, I've failed in scaring off people. So we'll see if we can do a better job. I think this particular lecture should do a very good job of that.

We're going to be getting into a little bit some more of the mathematics that underpins the associative array construct that is in the D4M technology that we're using in the class and gets into some group theory concepts that might be a long time since any of you had any of that or even recall what that is. It's actually not that complicated.

Personally, this is my favorite lecture in terms of it really gets into the deeper underlying elements. I would fully expect it to make your head hurt a little bit. We're going to do sort of a fairly rapid blast through the mathematics here.

And you won't really have to know this really to effectively use it. Although, it's good to know that it exists and that there might be pieces of it that you'll want to drive into deeper. And so really, the title of this talk is called Spreadsheets, Big Tables, and the Algebra Associative Arrays. We've lectured on that topic in the past.

So with that, we will get right into it. So here's the outline for the lecture. And I should say, you know, there's a balance between the lecture, and then we'll have some examples.

This particular lecture will be a little bit more heavy on the lecture side. And the example piece

is a little bit shorter. And so a little bit more lecture and a little less example on this particular class.

So we're going to talk about kind of a mathematical concept for what are spreadsheets. And in the introduction here, talk about what our theoretical goals are, really what associative arrays are. Getting into the mathematical definitions, what is the group theory? And then leading into the sort of traditional linear algebra, vector spaces and other types of things, sort of getting you a sense of this really is what connects those two things at the deeper mathematical level.

So what are spreadsheets and big tables? So spreadsheets-- obviously, Microsoft Excel tends to dominate the spreadsheet world-- are arguably the most commonly used analytics structure on Earth. Probably 100 million people, if not a larger number, use a spreadsheet every single day. Perhaps a good the fraction of the planet now has used a spreadsheet at some point or the other.

Big tables, which are these large triple store databases that we have discussed in prior lectures, are really what is used to store most of the analyzed data on Earth. So Google, Amazon, they all have these giant triple store databases, which can be thought about as gigantic spreadsheets really, rows and columns and values and stuff.

And the power is that they really can simultaneously store diverse data. You can hold in the same data structure strings, and dates, and integers, and reels, and all different types of data. And you can treat them very differently.

As we've shown here, this is just one spreadsheet. We can treat it as a matrix. We can treat it as functions, hash table, little mini databases. All in the same spreadsheet, we can kind of hold this information. So it's a very powerful concept, the spreadsheet.

Yet, we lack any formal mathematical basis for this. In fact, I looked up in the American Mathematical Association in the SIAM databases the word spreadsheet. And it did not appear in a single title or an abstract other than, say, maybe with reference to some software.

But this core thing that we use every day-- every single time we do a Google search we're using it, every single time we do a spreadsheet, we're using it-- has no formal mathematical structure. And so that seems like a little bit of a problem. Maybe we should have some formal mathematical structure.

And so this mathematical thing called an associative array actually gives us a mathematical structure that really encompasses this in a fairly nice way. I'm sure there's cases that it doesn't. But the core of what it is really, really works. So I think that's a powerful concept.

So what is our overall goal here? So we want to create a formal basis for working with these types of data structures in a more mathematically rigorous way. This allows us to create better algorithms. Because we can now apply the traditional tools of linear algebra and detection theory to these types of data.

It saves time in terms of, generally, if you have the mathematical background, you can implement things using a lot less code, which usually means a lot less lines. And often, you can build on top of existing optimized libraries, which give you better performance. And we throw out a number of 50 x less effort. For those with the required training, they can do this. And that is something that we've observed a number of occasions.

And you know, it's good for managers, too. I've tried to recruit group leaders at Lincoln Laboratory. It's like, you really should use D4M. I could connect it to SAP no problem. Get them feeling like they're actually doing programming again, you know?

I haven't had too many takers of that yet. So hopefully, maybe as you folks start using it, they'll see you using it. Binding to the various enterprise databases at the laboratory is very easy to do. And using D4M to analyze it is fairly easy to do.

So I actually use it now instead of-- if I'm going to use Excel, I'll often be like, no, I'll just write a CSV file and just use D4M for manipulating basic Excel types of operations. Because it feels a lot more natural to me. And so I certainly have done that.

So we're going to get into this associative array concept, which we discussed in some of the earlier lectures. We're going to get into it more deeply now. But the real benefit of the associative array concept is it naturally pulls together four different ways of thinking about your data into one structure. And you can apply the reasoning from any of these four ways of thinking sort of interchangeably now.

So we can think about data as associative arrays, as arrays that are indexed by words. This is similar to what Perl does in hashes of hashes, where we can have row keys that are words, and column keys that are words, and then values that are words or numbers. This concept is one to one with our triple store databases. So we can go naturally from triples to this and

inserting data into a database.

And if our data is an associative array, we can very quickly say, please insert into a database. And if we do a query, it can return an associative array to us very nicely. This obviously connects to graphs if we want to store relationships and graphs between vertices and entities. So we have alice, bob, cited, which is a duel to the linear algebraic matrix formulation of graphs. Row, Alice, column, Bob, their edge associated between them.

So we have these four very powerful ways of looking at our data. And they're now all sort of brought together as one concept. So I kind of argued it's one of the best deals in math you'll get. If you know one, you get the other three for free.

And that's really what the whole point of math is, right? If you give me a whole bunch of information, I can calculate something else that you didn't know. And typically, in signal processing that deal is more like, well, if give me 50 pieces of information, I can give you the 51st. Here, if you give me one, I can give you three. So it's a really good bargain.

As a little anecdote, I was working with my daughter teaching her the parallel geometry lines problem. You know, parallel lines. I was telling her this is the best deal in math. Because you know one angle, and you get the other seven free. So we're not quite as good as that.

That's probably the best deal in math going. And that's why they teach it so much. If you know one angle, you get the other seven for free.

Here, if you know one of these concepts, you essentially get the other for free. So it's a very powerful concept. And it's really kind of at the core of the benefit of this technology.

As much as possible, we try and make it that any operation on an associative array returns another associative array. So this gives you sort of the mathematical concept of closure. Linear algebra works because the vast majority of operations on a matrix return another matrix. And so you can compose these together mathematically and create sequences of operation, which are conceptually easy to think about.

So because of that, we can create operations like adding associative arrays, subtracting associative arrays, anding, [? warring ?] them, multiplying them together. And the result of every one of these operations will be another associative array. And so that's a very powerful, powerful concept.

And likewise, we can do very easy query operations if we want to get rows Alice and Bob, row Alice, rows beginning with al, rows Alice to Bob, the first two rows, everything equal to 47. And every single one of these will return another associative array. So again, this compositional concept is very, very important.

It's not to say we don't use the triples formulation, too. And we have routines that very quickly bounce you back and forth between them. Because there are times you're just like, no, I want to view this is as a set of triples. And I want to do new operations that way as well. So we support them both.

But this is really sort of the core composable aspect. In fact, most people initially develop their codes in this formalism, and then maybe change some of their lines to working on native triples if they need to get some improvement in performance, memory handling, or the side like that. And we certainly do that, too.

These associative arrays are actually very easy to implement. And the whole D4M library-- it's probably a little larger now, but when I wrote this-- it was about 2,000 lines, At which is very easy to do in programming environments that have first class support for two dimensional arrays, operator overloading. We're overloading operators here like crazy here, the plus, the minus, the and, parentheses, all that kind of stuff. And also, have first class support for sparse linear algebra. Internal, under the covers, we're using sparse linear algebra like crazy to make all this work.

Of the languages on Earth that have those features, MATLAB is by far the most popular one, which is why we have chosen that language to implement these features. There are other languages that have these features. And you can implement that.

But there are other languages that don't. And if you want to implement D4M in those languages, you certainly can. You just have to write more code. So this is sort of the language in which it can be done in the minimum amount of code.

And again, we find that for complex analytics that people typically can get their work done with 50 times less code than equivalent Java and SQL. And this naturally leads to high performance parallel implementations, because these are all arrays and matrices. And we have a very well-established literature in the community about how to do parallel linear algebra, how to make matrices work in parallel. It's a very well-studied discipline. And we have a lot of technology.

Just to remind people of how we actually store the data in our databases and how the data will often come out in our associate arrays, we use this exploded schema. So if we had data that looked like a standard table here with some kind of key and three columns here with different values associated with every one of them, if we were to insert these into a standard, say, SQL type table, we would just simply create a table like this in SQL. But if we wanted to look up anything quickly, we would have to create ancillary tables that would store indices to allow us to do things very quickly.

And if we wanted to add a totally new type of data, we would have to rethink our schema. So what we typically do is we create this exploded schema. Because our triple stores are very comfortable dynamically storing very large numbers of columns.

So we essentially typically take the type and append the value. So we create a series of triples here, usually with some minimal information for the actual stored value, something we would never want to look up. Again, by itself this doesn't give you much. Because most of these triple stores have an orientation, typically row oriented. So you can get any row quickly with relatively little effort.

And so we also store the transpose of that. We now create a table pair, such that we can look up any row or column very quickly. And so essentially, we've indexed all the data. And you can now do things very, very quickly.

So that's a very powerful, powerful concept. And we exploit it all time. And of course, this whole concept looks very much like an associative array. Mathematically, we can treat it as an associative array.

All right. So that's sort of the easy part. I think we've covered a lot of this before in previous lectures. Now, we're going to get into some of the more mathematical elements, which aren't necessarily hard. They just impose notation that we don't necessarily use in our regular day to day work lives. So we're going to talk about what are the values associations, keys, functions, and getting into things like matrix multiply.

So mathematically, we treat an associative array as sets of keys and values that are drawn from infinite strict totally ordered set, we'll call it S. You might be like, what is that? What is an infinite strict totally ordered set?

It's just a set where there's an ordering, where basically any two elements in the set, there's a function that will tell you one is greater than the other. And they also have an equal operation, too. So it will tell you that, you know, to give it any two values, I can ask the question is one greater than the other or are they equal? So that's the strict part.

So that encompasses a lot. Obviously, it encompasses the usual things like real numbers, and integers, and things like that. But it also encompasses strings if we impose an ordering on them. And we will choose lexicographic ordering almost always to be our order.

In fact, in the implementation, we sort of impose that. We always use lexicographic ordering. We don't really have another ordering that is readily at our disposal.

So given those keys and values, an associative array is a partial function from a d dimensional set of keys to a value. So if we have the vector of key, so typically d will be 2. And the first k1 will be the row key, and then a column key. And it will map out its partial function from the d keys to one value.

So it will look like this, an associative array with a vector of keys, typically 2. I will have a value vi and that it is empty otherwise. So this is its function, because we only define it where these keys exist.

So you can Imagine that every singe associative array is a function over the entire space of all possible keys. But we're only giving you the ones that are defined. And everywhere else, it's undefined. OK.

This is actually a fairly large break from linear algebra, where we can have functions-- a matrix is really a function of the indices, the ij indices, which are integers. Here, we allow them to be anything. And we formally support the concept of a completely empty row or column.

So associative arrays don't allow that. They only store not empty information. And in our implementation, we tend to store 0 as equivalent to null in this space. Because the underlying sparse matrix implementation does that.

0 is essentially treated as the null character. So that's a big difference. Now, we can still use our linear algebraic intuition, but that's the fundamental difference here.

So binary operations on any two associative arrays-- so I have associative array A1 and A1 with this binary operation-- are defined essentially by two functions. The sort of upper function

here says how we're going to treat the keys. So if we have two associate arrays, their keys, some of them can overlap. Some of them cannot overlap. And our function will either choose to look for the union or the intersection of those.

So for instance, if we do addition, that's most associated with unioning the underlying keys. If we do other operations, like and, that's most consistent with intersecting the two sets of keys. So obviously, if it's union-like, the result will always be non-empty. If the function is intersection-like, then there's a possibility you can do a binary operation. And the result will be an empty associative array.

So we have that choice there. And we choose union and intersection, because they're the ones that are-- you could do other functions. But they're the ones that are formally covered in set theory and keep us from having to relax this very general condition here.

We can have equality. So we can check for intersection. And so we're OK there.

So if we have an associative array, A1 with a set of keys here, ki, and a value, v1, and another associative array with an intersecting ki with v2, then A3 ki will be given by this function. It's either going to be the keys overlap. So if ki are the same, they'll overlap.

And so there you go. And likewise, with this. You have this choice. It could either be a union or an intersection. And then what we do with the values-- like, we now have the collision is what we'll call these. Two keys from two associative arrays, the row keys and the column keys are the same.

Well, now, we want to do something. We apply this function f, which we call the collision function. And that will determine what the actual result is, what the new value of A3 will be.

If one of these was empty, then obviously the union of this we don't even have to worry about it. That's kind of the way we approach this.

If there is no collision, so that means you're applying your function with this, then f is never called. So we always just do that or that. So if it's a union function, then you just get v. If it's an intersection function, then you would just get the empty set. And the underlying function never is called.

And you might be like, well, why would I care about that? Well, later on I'm going to get to deal with things like, well, what is the zero in this math? Or what is one in this math? And this will

allow me to sort of address some of those cases by saying, look, if there's no key intersection, we can do that.

So the high level usage of associative arrays is essentially dictated by this mathematics. And again, part of the reason we chose this very general definition is that, by definition, numbers, reals, and integers are all included here. We have not thrown them out. And we've also included strings.

I should say at any time we can say, look, our values are just real numbers. And then we essentially get the full power of linear algebra. But that's true of this sort of card that we always try and play last.

OK. We try and keep our mathematics as broad as we can. And if then there's particular instances like, oh, well, I need the values to be real numbers or complex numbers here, or integers, or whatever. We can always say, all right, the values are that. And now, we get a whole bunch of additional properties. But we try and build up the mathematics in this general way first, and then sort of play that later.

AUDIENCE: Question.

JEREMY KEPNER: Yes.

AUDIENCE: So the way you've-- the general definition of keys, that's more general than you would need to analyze spreadsheets. Wouldn't it be sufficient for spreadsheets to have your keys be integers [INAUDIBLE]?

JEREMY KEPNER: So I believe that actually spreadsheets store things as R1, and R2, R3 internally and C1, that they actually use a internal triple representation, and then project that into this integer space. So what the actual spreadsheet does, I think, kind of depends on the specific spreadsheet. So the question was-- they can't really hear you on this mic.

Is this more than you need to do a spreadsheet? Could you get away with just integer indexing? And I would say perhaps. I don't really know.

And as I said, I think internally they actually do something that's more akin to this. Certainly when you do math in Microsoft Excel, it has letters. You give it A1, B1, those types of things.

AUDIENCE: But that's just because they're using letters instead of a second integer.

JEREMY KEPNER: Yeah. Yeah. One could view it that way. But people like it. So we do what people like, right?

AUDIENCE: Yeah. But you don't need an additional mathematical structure. [INAUDIBLE]. You can just make it one to one [INAUDIBLE] letters to--

AUDIENCE: [INAUDIBLE] like in Excel, you have different worksheets, too, and different columns. So you have to reference both of those.

AUDIENCE: You have to reference row and column. But you just need a pair of integers.

AUDIENCE: Well, it's a row and column. But then it's also the worksheets that you're on.

AUDIENCE: OK. So, three.

AUDIENCE: So there's three. But they're not going to sequentially number all those. Because it's a sparse space that you're using, right, probably? So I'm sure they do some caching. I don't think they store a number for every column that you're not using.

AUDIENCE: Certainly, you can try that out by creating an empty spreadsheet, putting the number in one corner, putting a number in another corner really far away and see if it [INAUDIBLE] two [INAUDIBLE].

JEREMY KEPNER: It doesn't do that. Obviously, it does compress.

AUDIENCE: [INAUDIBLE].

JEREMY KEPNER: Yeah. So mathematically, it may be more than what is minimally required to do a spreadsheet. We will discover, though, it is very useful and, in a certain sense, allows you to do things with spreadsheets that are hard to do in the existing technologies, which is why I use it.

There's operations that I can do with this that are very natural. And the spreadsheet data is there. And I would like to be able to do it.

AUDIENCE: So Jeremy?

JEREMY KEPNER: Yes.

AUDIENCE: The actual binary operator plus in associative array, is it union or intersection?

JEREMY KEPNER: So on the keys, it will be union where the value is an intersection with an empty set. We don't care what the actual numerical thing is. We'll just return the other thing.

Where there is a collision, and so now you have to resolve-- and that's kind of why we get a lot of mileage here. Because most of the time, we are very few collisions. And so it's not like we have to worry about huge things.

So where there are collisions, if the values are numbers, then it'll just do the normal addition. If the values are strings, then it's somewhat undefined what we mean there. And I forget if D4M throws an error in that situation if the values are strings.

It might default to just doing something like taking the max or the min and saying, you tried to add two things for which the formal addition collision function doesn't really make sense. And I'm just going to put something in there. We can test that out at the end of the class.

AUDIENCE: So a collision function is a function of that type of the value?

JEREMY KEPNER: Yes.

AUDIENCE: OK.

JEREMY KEPNER: It has to be a function of the type of the value. Because if I'm going to add a real number with a string, what do I do? Because I can do that. Yes, another question.

AUDIENCE: To follow up on the previous point, it seems like in the case for addition, for example, there might be cases where you'd actually want union, then add, or intersection, then add. It kind of depends on what the nulls in your array actually represent.

JEREMY KEPNER: Oh, there are so many functions where you can think, well, I'd kind of like to have it mean this in this context.

AUDIENCE: Yeah. So the question is can you pass in as a parameter in D4M I want you to union, then operate, I want you to intersect, then operate? Or did they all just assume the most logical and do it?

So bigger lever question was sometimes you maybe would like plus to mean some of the other conceptual ways that we would mean it. OK. So in fact, we see that a lot. There's a lot of things.

Now, in fact, we'll get into that space. And there's hundreds, thousands, millions of potential possible algebras that you can possibly want. So what we have done is we think about the

function. And we think about in this mathematical context.

And we see what its mathematical implications are in terms of is it consistent with the overall group theory of the mathematics, in which case we know that if someone uses it that the intuition will carry through? Or is it kind of forking them off into it's something you might want to do, but really the result is now taking you into a sort of undefined space? So that's kind of a thing we use for doing that.

We don't have the formal support in this of a multiply operation that allows you to pass in an arbitrary function for doing that. I should say there are various tricks that allow you to do that in just a couple lines or two. Basically, what you can do is you start with give me the intersection of the unions first. Now, I know I have only collisions.

Now, give me the values. Apply whatever operator you want. Stuff the values back in, and you're done. And so you can kind of work your way through that.

And by doing that, you're doing it in a way such that you know, yeah, you're outside of an algebra here. You're just kind of doing what you're doing. So encourage people to do that. But I'm just telling you in terms of how we choose what functions we pull and formally support.

And there's a few that I've been wrestling with for a long time, for a very long time and just haven't been able to decide what to do. Maybe I should just make a decision. Yeah.

AUDIENCE: Like a real simple example, if I had an array, one, two, three, and ray b was null on all three, there would only be one collision. And if we were in add, it would only add them at the last--

JEREMY KEPNER: No. If it was an addition, it was--

AUDIENCE: So one, two, three. And then the other one was null, null, three.

JEREMY KEPNER: So but they have the same column or something? These are vectors?

AUDIENCE: Yes.

JEREMY KEPNER: These are row vectors? Yes. So basically we have a one column vector with three, another column with one. And when we add them together, you would get a three. Because it would be union.

It would union the keys. And then the only addition operation would be performed on the

collision. We did one or and an and operation, then the result would just be one, an associative array with one element in it.

AUDIENCE: And we can do either one depending on the context? I kind of lost you there. Because the question was is it union or intersection.

JEREMY KEPNER: It depends on the operation. So plus formally is closer to set addition, and so therefore, is a union operation. And is formally closer to set intersection, and so it's an intersection operation there. Regardless, in both cases, the collision function is only applied where there is a collision. Yes.

AUDIENCE: I'm getting a little confused by [INAUDIBLE] formally defined operations on associative arrays in terms of values. You've used a quality on keys at this point. You haven't defined any other operations such as [INAUDIBLE] intersection [INAUDIBLE].

JEREMY KEPNER: Well, so the keys are part of a set. And so they get the union and intersection properties of strict totally ordered set, which is the intuitive use of union.

AUDIENCE: Right. But they belong to the same set, [INAUDIBLE] the same [INAUDIBLE] set. Yet, you've only defined binary operations on associative arrays. You haven't defined them on the elements of the set.

JEREMY KEPNER: So we'll get into that a little bit later.

AUDIENCE: That's what I thought.

JEREMY KEPNER: But they essentially have the binary operations of a strict totally ordered set, which is equality and less than.

AUDIENCE: It seems to me the bulk of the questions were about manipulations on keys we haven't even talked about yet.

JEREMY KEPNER: Yes. Yes. Well, there's another 40 some slides. But actually, you've gotten a lot of the good questions here. So the thing though to recognize is that there's a lot of choices here. And the algebra you are in is determined by the function that you choose.

And actually, you'll find yourself switching between different algebras fairly frequently. That's how we use spreadsheets. Sometimes we do this operation, and now we're in this algebra. And we do this operation, now we're in this algebra. And so that's the big thing to kind of get

away there.

So let's get into this a little bit more. That's sort of the big overall. So I think we've talked about this. Let S is an infinitely strict totally ordered set. Total order Is an implementation, not a theoretical requirement.

So the fact that we were imposing this infinite strict-- that they're totally ordered, that we don't just allow equality, is more of an implementation detail. It is very useful for me to internally store things in order so I could look things up quickly. However, mathematically, strictly, we just had a test of equality, all the math hangs together there. All value's and keys are drawn from the set.

And the allowable operations on them, of two values or keys, is less than, equal to, or greater than. So those are the three functions that we essentially allow. Strict totally ordered set only is two, which is less than, less than or equal to. But you get essentially the third one for free.

In addition, we have the concept of three special symbols here, which we've talked about, which is the null, the empty set, a least element in the set, and a maximal element to the set. So v less than or equal to plus infinity is always true. And plus infinity is, we're saying, the maximal element is a part of this set.

Greater than or equal to negative infinity is always true. In all set theory, the empty set is a formal member of all sets. So now we'll talk about-- did that sort of get to your question about the operations that are defined on the keys?

AUDIENCE: Yeah. Yeah. You [INAUDIBLE]--

JEREMY KEPNER: Right.

AUDIENCE: [INAUDIBLE].

JEREMY KEPNER: Right. So then we talked about the collision functions. And there are two sort of contextual functions here, union and intersection. Because there's the two operations that makes sense. And then three conditions, less than, equal to, or greater than. That is in order to preserve this strict totally ordered set.

Once I go to values that are reals or integers, I have more operations. But if I'm just limiting myself to values that are members of a strict totally ordered set, this is all I have. So that

means that you have d plus 5 possible outcomes to any collision function.

That is, you could have the collision function actually choose to put out the value of its underlying key. You could produce v1 or v2 or empty or minus infinity or plus or sets of these. So it's a fairly finite choice of results.

If we're going to stay in this sort of restricted thing, we have a very limited number of things that can actually happen. When you're actually applying a collision function on a particular where there's an intersection, you have a fairly finite number of things. However the total number of combinations of these functions gives a very large number. And their function pairs gives a very large number of possible algebras.

I did a back [INAUDIBLE] calculation said it was 10 to the 30th. It might be even more that. And in fact, it might be formally infinite for all I know. But that was an easy calculation. There's a lot of possible algebras.

And this is a fairly impressive level of functionality given our relatively small numbers of assumptions that we have here. But we are going to focus here on just the nice collection functions, the ones that seem to feel like they give us intuitively useful things. So we are not going to use keys as outputs to our function here. OK.

We will in some contexts. And we're going to say the results are always single valued. We're not going to deal with results, expand our values to be sets of values. Although, in certain contexts we will do that. But mathematically, let's restrict ourselves to that.

We're going to do no tests on special symbols. And we're just going to basically say our collision function can be essentially-- if we have v, less than v2, the answers can be one of these five. If it's equal to these, it can be one of these four. If it's greater than these, it can be one of these five.

So it gives a fairly limited number of possible collision functions. And all these properties are consistent with strict totally ordered set. And generally, when we handle a value of this, it's handled by the union and intersection function first. It never actually gets passed through to this function.

All right. So let's move on here. And just as I said, well, what about concatenation? In fact, there are contexts where we will want to concatenate a couple of strings together or something like that.

And you know, that's fairly supported. I think the results can be the sets themselves. You would have a new special symbol, which is the set itself.

You have now different collision functions. So you have union. And then I want a union values as our intersect, and then union values and union and intersection. So those are your functions there.

And there's actually a few instances. I think we've already showed one of them in one of the examples where we actually do this. So I'm just sort of throwing that in there that concatenation is a concept that we do think about. I don't think we've developed the formalism as richly in a situation where the values are simply single values. But we certainly can and do do that.

AUDIENCE: Jeremy.

JEREMY KEPNER: Yes?

AUDIENCE: Third example, was that [INAUDIBLE] for identity instead of union? [INAUDIBLE].

JEREMY KEPNER: This one here?

AUDIENCE: No, go up. Get a little bit to the left. The symbol-- [INAUDIBLE] was upside down.

AUDIENCE: I have a similar question. Is the one above it [INAUDIBLE].

JEREMY KEPNER: Yeah. I might have a typo here. I should double check that. Yeah. So v union that. All right, so that should be an intersection.

And that should be union. And that should be intersection. And that should be union.

So I just got them to flop there. So thank you. Full marks to you. We will make that change.

All right. So that one is intersection, correct? And this

AUDIENCE: [INAUDIBLE].

JEREMY KEPNER: --three should be intersection. All right. Very good. We'll check in the SVN. In the next down time, it will be part of your [INAUDIBLE]. Moving on.

All right. So one of the things we're eventually going to work towards is trying to make matrix

multiply work in this context. OK. We've already talked about the duality between the fundamental operation of grass, which is breadth first search and vector matrix multiply. We continue to want to be able to use that. It's a very powerful feature.

And what we find is that in graph algorithms, where we're dealing with things on strings or other types of operations, that most graph algorithms can be reduced to operations, what are called on semi-rings. So those are generalizations of normal linear algebra, whereby your two core operations here, what is called the element wise multiply and the element addition operation, are this has the property of being associative and distribute over the plus operation. And plus has the property of being associative and commutative.

And examples of this include the traditional matrix multiply plus and multiply, min.+, or max.+. In fact, there's a whole sub branch of algebra called max plus algebras, which do this. Another one is or.and and other types of things. So mathematically, semi-rings are certainly well-studied more in the context of specific algebras, like min.+, min+, or max+, or something like that.

Here, with the associative arrays, we're dealing with you could be hopping and popping back and forth within the same expression. You could be moving back and forth between these different algebras. So I think that's a little bit different.

We're taking sort of a bigger view, a more data-centric view. Our data can be of different kinds, and then impose upon it different algebras as we move forward. And the real theory questions that we're trying to answer here is we have this concept of associative arrays, which is kind of new. We have the traditional linear algebra. And there's going to be areas where they overlap.

Your ideas, your intuition from linear algebra, and your intuition from associative arrays will be very, very well-connected. OK. And there's going to places where associative arrays give you new properties that you didn't have in linear algebra. And those will allow you to do new things. And there's going to be cases where your linear algebra intuition is wrong. That is, taking your linear algebra intuition and applying it to associative arrays will lead you into things that won't make sense.

And so this is kind of what we're trying to give people a sense here, of where they overlap, where you should watch out, and where there's new properties that you didn't know about that

you should maybe take advantage of. The biggest one being the universal conformance of addition and multiplication of all associative arrays. So any two associative arrays, regardless of their size, can be multiplied and added to each other unlike traditional linear algebra, where you have very strict constraints about the dimensions of matrices in order for them to be added.

And we do use this all the time. We use this all the time, essentially almost exactly in the example that we talked about. It was a question that was brought up earlier. We can add a vector of three elements and a vector of five elements and have something that makes sense.

OK. Likewise, we can multiply a 10 by 12 with a 6 by 14 and have something that actually makes sense. So that's a very powerful concept. And I'd say it's one of the most commonly used features.

All right. So we've done the basic definition. Now, we're going to kind of probably move a little bit quicker here into the group theory. So the group theory is when you pick various functions. What kind of algebra's a result of that?

So we're going to talk about binary operators, and then get into commutative monoids, semi-rings, and then something we call the feld. So our joke there is that so a field is when you do the traditional mathematics of linear algebra. You're doing them over the real field or the complex field of numbers. It's a set of numbers with certain defined properties. OK.

When we impose this condition that we are doing with strict totally ordered sets, we don't have an additive inverse. There's no way to formally subtract two words from each other. We can't have a string and a negative string.

So we don't get, like, a and minus a, which is part of a field. So we have is feld, which is a field without an inverse. So mathematicians are very funny. And when they do this, they often will drop the i or something like that to make that concept.

So we will be doing math over a-- we can have much of the properties of linear algebra. We just have to recognize that we don't have additive inverse. So it's sort of like vector spaces over felds. Or sometimes they're called semi-vectors spaces. You can put semi in front of anything. And it's like it's a little bit of this, but minus something.

So this is kind of our operator roadmap. So you see here, we began with three definitions. We will have narrowed it down to 200 operators of interest of those nice collision functions and

union functions. There's about 200 combinations that work well.

We will find that 18 of those operators are associative. So not associative in terms of associative array, but associative in terms of the formal mathematical concept of associativity. And so those form what are called semigroups.

14 of them are commutative, which means they are Abelian semigroups. Commutativity, that just means that you can switch the order of operations. a plus b is the same as b plus a. Not something that is a strict requirement. But gosh, it's sure nice to have to not have to be like oh, no, I switched the order. And now the answer is different.

So traditional matrix multiply is like that, right? a times b is not the same thing as b times a. And we will have that as well. But commutativity is really nice.

We then take these 14 operations and explore all their possible pairs. There's 196 of them. We look at which one of them distribute. There's 74 that distribute, so they form semirings.

We then look to see if we have an identity and an annihilator. Some of our special symbols, can they perform that role in which case they can form a feld? And then we can create, essentially, vector semi-spaces or vector spaces over these felds. And we'll have about 18 of them when we're all done.

And the good thing is when you get all done with this, all the ones that you like are still in the game. And a few of the ones you might like, some of my favorites, they don't get as far down this path as you want. But then you have these properties. And you're like, OK, I have vector space properties and other types of things.

And I'm sure some of you will find issues or comments or criticisms that we haven't written this up formally in journals. I've been desperately trying to hire abstract algebraists as summer students and have not been successful so far. So if you know any abstract algebraists that would like to work on this, we would be happy to pay them to help us work on this.

If we extend our definition of values to include sets, ie, to include concatenation, this is essentially what it does. It adds four functions here. They actually make it through all these operations here.

And they basically keep on going. But then they fall away here when we try and get to this final feld step. But you know, this is still a very useful space to be in.

All right. So let me explain a little bit of how we sort of navigate this space. So if we can limit ourselves to special function combinations that are associative and commutative-- so just remember, associative just means you can group your parentheses however you want. I always have to look it up.

Never once been like associative-- I mean, I don't know. It's never been intuitive to me. But that's the grouping of parentheses.

Commutativity just means that you can flip. So these are our functions. So these are all the 18 here. And then the grade-- one's grade out of the ones you lose, because of the commutativity. So basically this function, which is just left, which just says return the left value in all circumstances, well, obviously, that's not commutative.

And quite frankly, left and right are sort of silly functions. You Sometimes it almost is essentially a no op. If you knew you want the left value, you just take the left value and move on.

So that doesn't really-- so then we're left with a lot of ones we like, essentially union max. So basically, take the max value, union min. I call this the intersection delta function just return. It only gives you an answer if the values are the same.

This is essentially the union. It's sort of like the x or function. If there's a collision, blow it away. And then there are very sort of other kind of more unusual combinations here dealing with the special functions.

We tend to kind of really live mostly here. These tend to be the ones that really are the ones you use a lot. I haven't really used these too much.

All right. And so we're left with these, what are called, Abelian semigroups are these 14 highlighted ones here.

AUDIENCE: So Jeremy, I feel I'm going to get lost here. What does Abelian mean?

JEREMY KEPNER: So Abelian is a semigroup that is commutative. That's all it means. So basically, it means you have associativity.

And then it's Abelian if you add commutativity to it. So any sets of numbers that obey grouping operations and are commutative is Abelian. I mean, I think Abel did a lot more that he didn't

need to get his name associated with that. It could have just been called comm-- it's sometimes just called a commutative semigroup.

So you know, it's lucky when you get your name added to some really trivial operation. I mean, he did a lot of things. But they threw him on here.

And in fact, in group theory, they complain about this. And there are numerous rants about how they've named all their groups after people, and they don't tell you anything. As opposed to other branches of mathematics, who would simply just call this a commutative semigroup or just a commutative associative group.

That might be even more-- and of all the different properties, most of them you can combine in whatever you want. There's almost an infinite number of groups. So there's certain ones that are most useful. Let's see here. So now, we start--

AUDIENCE: Along those lines, what's the-- and I'm sorry if you already said this-- distinction between a semigroup and a group?

JEREMY KEPNER: A group and a semigroup. I want to say-- oh, it's an Abelian group without inverses. So a semigroup there. So these are all Abelian semigroup or Abelian groups without inverses or commutative associative groups without inverses.

So you see the problem. And so I guess, he studied them a lot. So he got his name associated with them and proved their properties and stuff.

So we have 14 of those that form 196 pairs. So these will begin to-- so we want to look at the ones that are distributive. So that basically means we assign one of those operators to be the addition operation, the other to be the multiplication operation.

And we need to show that it is distributive. And of those 196, 74 operator pairs are distributive. These are called semirings. Or they could be called rings. So a semiring is a ring without an inverse, so rings without inverses and without identity elements.

And if you look at the various definitions, I mean, there's the Wikipedia definition, which mathematicians will say is that's not really a true definition. There's a Wolfram math rule definition, which is more rigorous. And they often disagree on this stuff.

I tend to be like if Wikipedia says is true, then most people on the planet Earth believe that that

is what is true. And so therefore, you should be aware that is the truth that most people will believe. So I don't know which one we've chosen here, whether it's the Wikipedia or the Wolfram. There's also various encyclopedias that define this stuff, too.

AUDIENCE: Jeremy?

JEREMY KEPNER: Yes.

AUDIENCE: Can you define ring?

JEREMY KEPNER: Define what?

AUDIENCE: Ring.

JEREMY KEPNER: Ring?

AUDIENCE: Yeah.

JEREMY KEPNER: Oh, uh, well, a ring would be this. And it would have inverses and identity elements. So it's basically something that satisfies that's distributive and has inverses and identity elements.

But if you type ring in the Wiki-- and I should say, the great thing about Wikipedia is like all the definitions are linked. So after seven or eight clicks, it kind of all holds together. So it's pretty nice.

And Wolfram is the same way, just enough knowledge to be dangerous. The internet's your friend, right? So moving on here. And this is something that's less important, but kind of for completeness if we're going to head towards vector space, we need to address, which is the concept of identity elements.

So zero is the additive identity. When you think of normal math, you add zero to something, it doesn't change it. And the choices for additive identity elements, we have three special symbols, and we could pick them.

We have the multiplicative identity, which is 1, and the multiplicative annihilator. So of the choices here, we have 12 semirings with the appropriate zeros and ones. We have four that actually have two combinations.

And we have 16 total operations-- of the 16, there are 6 operators. These are different operators. And again, we call these felds without inverses. I'll get into it a little bit more.

So for instance-- and we can skip that for now. So just a better way to look at that is these are our operator pairs. OK. And we wanted to see which one of the them sort of form these felds.

So the ones that distribute are marked with a D here. The ones that distribute and have a 0, 1 operator pair that works, are shown in the square here. And some of them have two.

So if I pair [INAUDIBLE] plus union min and multiply a intersection min, and I define 0 to be the empty value, and 1 to be plus infinity, then I can create a feld, essentially, out of that. I had a lot of debate with someone about this one, whether I can have plus infinity be the 0 element and have it be less than the 1 element. Was there a requirement that 0 actually be less than 1 in this definition?

When I talked to mathematicians, they're like, eh, it's kind of what you want, then you make it that. So this just shows the full space of things that are possible. If we go back to our concatenation operators here, so these were our four concatenation operators and our collision functions. And this shows you what that set looks like.

So these are the four by four pairs here. All operators, they all distribute. And 16 of these form semirings. Because you're able to construct these various zeroes.

And this isn't, like, rigorous. I mean, I might have messed one up or a few up here or there or something like that. But this just kind of gives you a sense of the space that we're working on.

AUDIENCE: You made some typos on that table [? you ?] fix later.

JEREMY KEPNER: Probably. Probably. Now, I'm going to kind of really move it forward here. So we get into vector spaces.

So we can have associative array vector addition. Again, all associative arrays are conformant. We have the concept of scalar multiplication, which is essentially applied to all values.

So one of the things I really struggle with-- so scalar multiplication kind of makes sense, right? If I have an associative array and I multiply it by a scalar, I can imagine just applying that in an intersection sense only to the keys are defined by one. Scalar addition, though, is very difficult.

If I have a scalar plus an associative array, does it only apply to the keys? Or is a scalar really the associative array that's defined everywhere over all things? So it's infinite. So that's something I struggle with.

So when you ask why is scalar addition not supported in D4M, it's because I don't know what it's supposed to mean. And you can easily just pop out the values, add whatever you want, stuff them back in, and can be on your way. And then your safe.

So in the vector space that we form, it meets the plus requirements. It commutes. It's associative. We have an identity.

But we have no inverse. So we have to be careful. That's why we don't have add.

And a vector space, it meets the scalar requirements. So all associative array operator pairs that yield felds, also result in vector spaces without inverse spaces. Maybe we call these vectors semispaces, I don't know, or vector spaces over a feld or something like that.

What kind of properties here? Well, we have scale identities. That's great. You could create subspaces. That makes sense, too.

The concept of a span, yes, you can definitely do concepts of spans. Does span equal a subspace? So this is a big question in vector space theory, spans on subspaces. Not sure.

Linear dependence. Is there a nontrivial linear combination of vectors equal to the plus identity? You really can't do this without an additive inverse. And so that becomes a little bit-- so we really kind of need to redefine linear independence, which we can do. But there's a lot of the proofs of linear independence and dependence that rely on the existence of inverses. And you probably could circle your way around that.

AUDIENCE: Question. Are you just missing the additive inverse or both the additive and the [INAUDIBLE]?

JEREMY KEPNER: Both. Both, yeah. Yeah. So one of the things is considering a linear combination of two associative array vectors. Under what conditions do they create a unique result? So this really depends on what you choose.

So for instance, if we have a vector A1 and A2, when we multiply it by coefficients A1 and A2, and we use these as our plus and our multiply, and this is our 0 and 1, where are A1 uniquely determined? And so for instance, if I pick, in this case, our canonical identity vectors, which is A1 is equal to infinity and minus infinity here, then we find that we can cover the space very nicely. If we do it the other, we can't.

A better way to view that is in the drawing here. So here's my whole space of coefficients A1

and A2. And we see that A1 and A2 uniquely define a result with these basis vectors. And A1 and A2 are completely degenerate with these basis vectors.

So depending on the kinds of basis vectors you have, you can create unique stuff or not unique stuff. You can give yourselves actual values here. Basically, if A1 is just equal to this value and A2 is equal to this value, this shows you what that looks like.

There's places where it's unique. There's places where they're the same. And there's places where one is unique, but the other is not unique. So again, you can construct using associative arrays as basis vectors, a very rich set of things.

And the same thing goes with multivalued vectors. Again, different types of spaces here. We really need to kind of work this out. If anybody's interested, we're very interested in having people help us work this stuff out.

Which of these operations make sense? Transpose makes total sense. Transposing of associative arrays makes total sense. And it's a very efficient operation, by the way, in D4M. You can do transposes very efficiently. It works out very nicely.

Special matrices, submatrices, zero matrices, square matrices, diagonal matrices, yes. Although, diagonal matrices are a little bit tricky. Upper and lower triangular, yes, you can kind of do this. Skew symmetric, no.

Hermitian, not really. Elementary and row column, sort of. Row column equivalence, sort of, under certain conditions. These are all things you can do in linear algebra. And sometimes you can do them with associative arrays and sometimes you can't. You have to think about them.

Matrix multiply is sort of our crown jewel. Always conformant-- can multiply any sizes whenever you want. There's two ways to think about this. You can make your head hurt a little bit when you start dealing with the no elements.

When does the union operator get applied? And when the-- so when you do computation, there's two ways to formulate a matrix multiply. There's the inner product formulation, which is typically what people use when they actually program it up. Because it tends to be more efficient. That's basically you take each row and each column, you do an inner product, and then do the result. OK.

Mathematically, from a theory perspective, you get yourself in less trouble if you think in terms

of the outer product formulation, which is basically you take each row and vector, you do the outer product to form a matrix, and then you take all of these, and then combine them all together with the operation. And that, theoretically, actually keeps you sane here. And that's the way to think about it mathematically.

Variety of matrix multiplies examples, I won't go into them here. Obviously, they depend heavily on what our collision function, g, is here. It gives you different values and different behaviors.

The identity element, maybe a left identity, right identity. In some instances, it seems to be OK. But the identity [INAUDIBLE] is a little bit tricky.

Inverses, boy, is it hard to construct inverses when you don't have underlying inverses. I It's just really tricky. And so, probably are not going to get anything that looks like an inverse.

You can do Eigenvectors in certain restrictive cases sort of. And there are interesting papers written about this. But it's only on very-- the row and column keys need to be the same and stuff and so, you know.

One thing I really would like to explore is the pseudoinverse. So pseudoinverse A plus satisfies these properties. And I actually think that will are probably be in pretty good shape for pseudoinverse. And the pseudoinverse is what you need to solve the least-squares problem.

And I think solving the least-squares problem is actually something we might really be interested in doing in some of our problems. So I do need to work this out if people would like to explore this with me.

We have a whole set of theorems that we'd like to prove. Spanning theorems, linear dependence, identities, inverses, determinants, pseudoinverses, Eigenvectors, convolutions, for which of these do these apply? A lot of good math that could be done here. Call to arms for those people who are interested in this type thing.

So just to summarize, you know, the algebra of associative arrays provides us this mathematical foundation. I think I've have tried to show you the core parts that are really well solid and expand that to the points here. You can see where it is, while we don't really know exactly what should be happening here, and give you a little bit a logic behind how we do this. A small number of assumption really yields a rich mathematical environment.

And so I have a short code example. It's not really teaching you anything new. It's just to show you that I tested all these properties using D4M, which was nice and really in a very kind of spreadsheet kind of style. And so I'm just going to show you that. There's really just one example. It takes a few seconds to run.

And then the assignments-- so these are in this part in the directory-- And then the assignment, should you so do it-- if you didn't do the last assignment, well, you're going to need to do that to now do this assignment. So basically, for those of you who did the last assignment-- array of a drawing and looking at the edges and stuff like that.

Now, I want you think about that associative array. And think about which kind of these operations would make sense if you were to try and add them or multiply them or whatever. Just explore that a little bit. And just write up, OK, I think these kind of operations would make sense here.

You know, addition would make sense if it's union and the collision function is this or something like that. You'll have to think about what your values are. It could just be your values are just 1, something like that. You might be like, oh, my values are 0,1 and I want 1 plus 1 to equal 0. So you might have an "or" or "x or" operation or something like that.

So just sort of think about what your example was. And think about these ideas a little bit. And just write a few sentences on kind of what that means.

The last slide here is-- we don't really get this question. But if you want to compare what the difference between associative rays are and the algebra defined by Codd that sort of is the basis of SQL, there's this little table that describes some of the differences there.