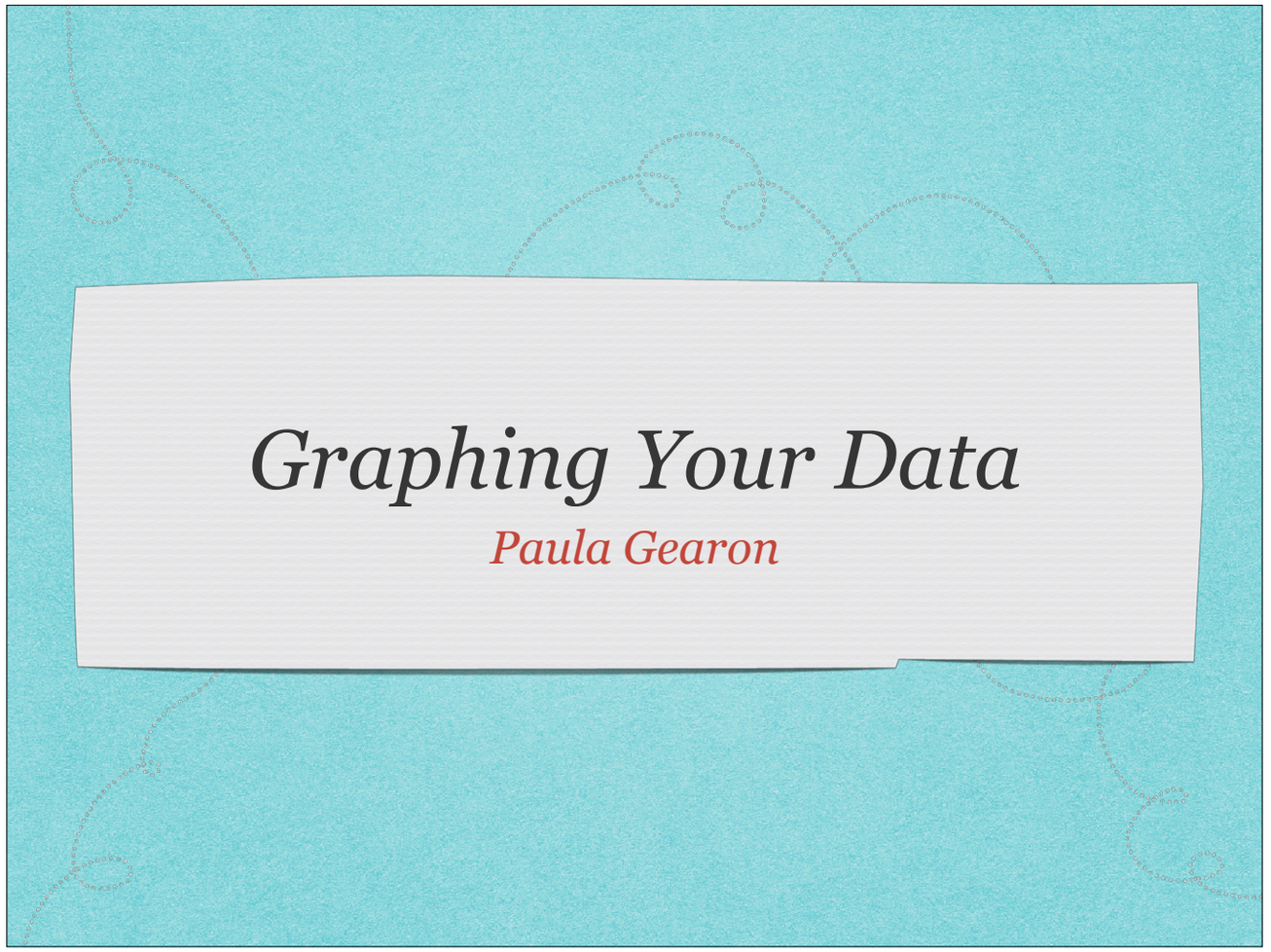


Hello everyone.
My talk today is about graph databases. I've been working on, and using various graph databases for about 18 years. Today I'll be explaining a little bit about how they are used, and showing some of the processes of how they work.

Many Database Types

- SQL (RDBMS)
- Document
- Key-value
- Graph

There are a number of different types of databases out there. The most common are relational databases, which we also refer to by the name of their usual query language: SQL. Then there are databases that use other semantics in their structure, and we often lump all of these together with the label NoSQL. These include Document oriented databases, Key-value stores, and graph databases.

Graph Databases

- RDF/SPARQL
- Neo4J
- TitanDB
- Datomic
- Datascript
- **Asami**

Within the family of graph databases, there are a number of styles and implementations. Regardless of these difference, the semantics of the graphs stay mostly unchanged, meaning that the APIs and query languages for dealing with them usually follow the same ideas.

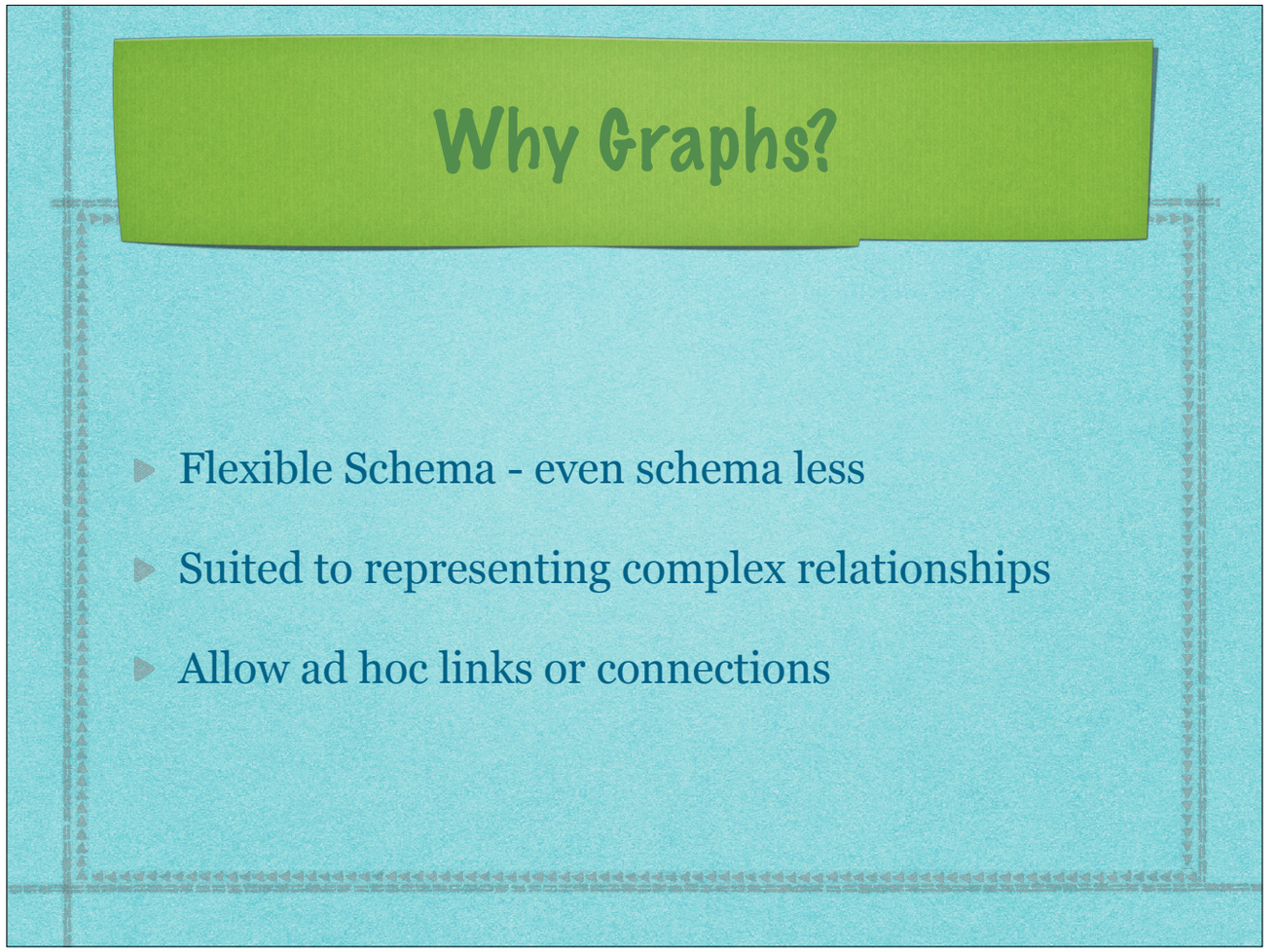
The first ones I'll mention are the RDF/SPARQL databases. These store data in a datamodel defined in the Resource Description Framework, or RDF, and they implement all the standards defined in SPARQL, including a query language, and communication protocol. Both RDF and SPARQL are W3C standards. These formalize a number of the concepts of dealing with graphs, with some of the ideas of the web mixed in. This leads to the name "Semantic Web". There are a number of open source and commercial RDF databases out there, some of which I have worked on, like Mulgara, Jena and Sesame.

There are a number of other databases that don't follow specific standards. One of the most popular is Neo4J, and many people working with graph databases may have encountered this one. Titan DB is an open source database supported by some large companies, like IBM, for providing cloud scale graphs.

Of special interest to us, are the databases that came about with Datomic. Datomic is built using Clojure, and the APIs for communicating with it are all native to Clojure. It's built over persistent data structures, similar to those in Clojure, which doesn't actually make it unique, but it provides an API that gives developers access to a full historical timeline of the data, which is new.

Datascript is an attempt to build an open source "Datomic Light", which holds a non-durable version of a database, and can be incorporated into JavaScript apps as a library. Asami was a project that I started at almost exactly the same time, for similar reasons, and I didn't learn about Datascript until about 18 months later.

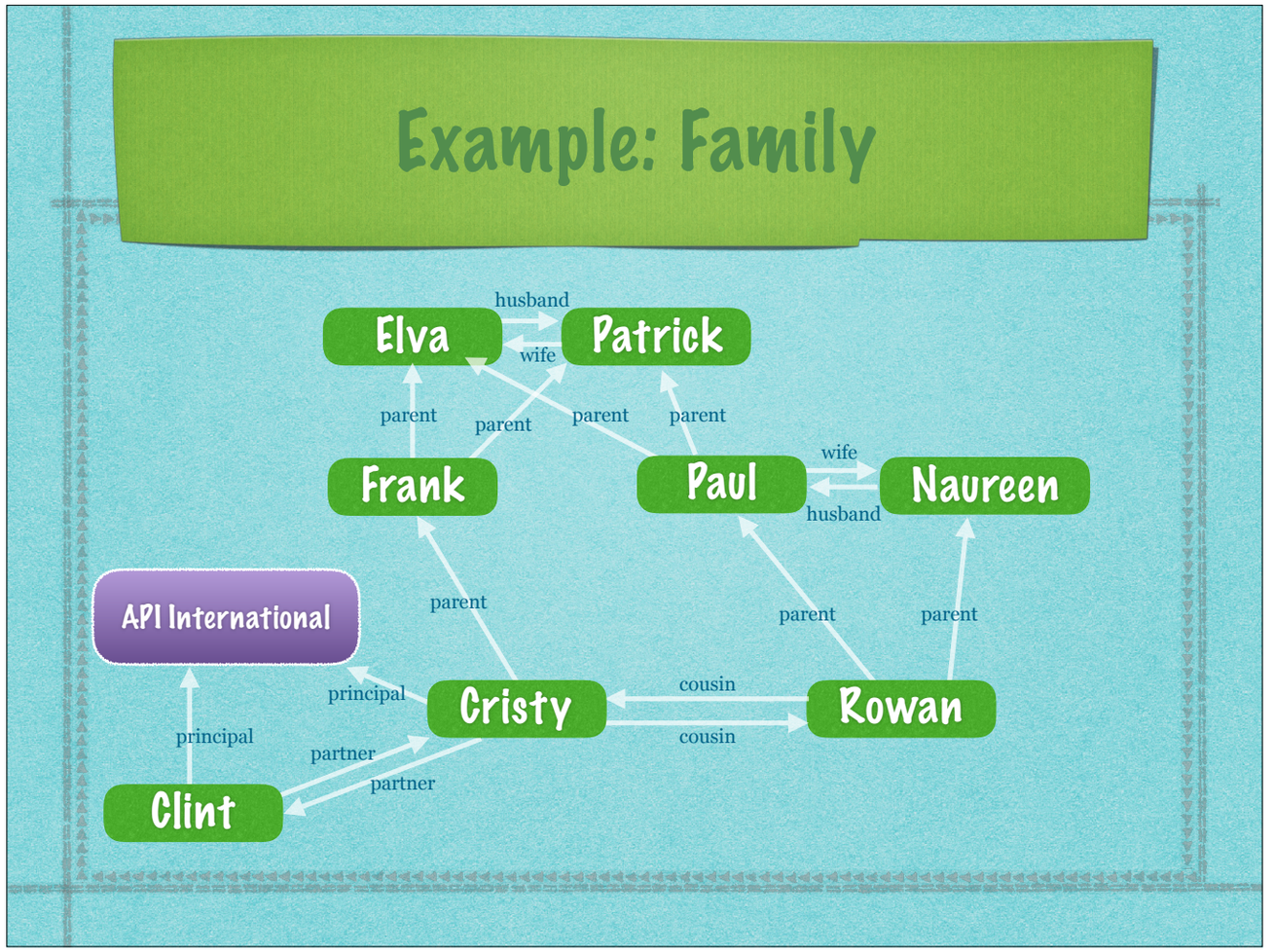
* One difference between Asami and Datascript is that Asami has simpler storage indeces, which makes it easier to discuss in a talk like this, so I will be focusing on its implementation. However, almost everything here will be applicable to the other stores as well. It's these similarities that I'm hoping to get across today.



Graph databases stand out in a few ways for the user. They have very flexible schemas, and many can even let you operate without any schema defined for the system at all. This is a bit like using Clojure maps vs records. Records define what you expect to find in a data structure, whereas maps operate almost the same without having any expectation of what will be in them. Most of the time your code still needs to know what's in there, but you have more flexibility when there is nothing defined up front.

Because they represent data as a set of connections between simple things, graphs can actually represent any kind of data, and are composable, making it natural to deal with complex structures.

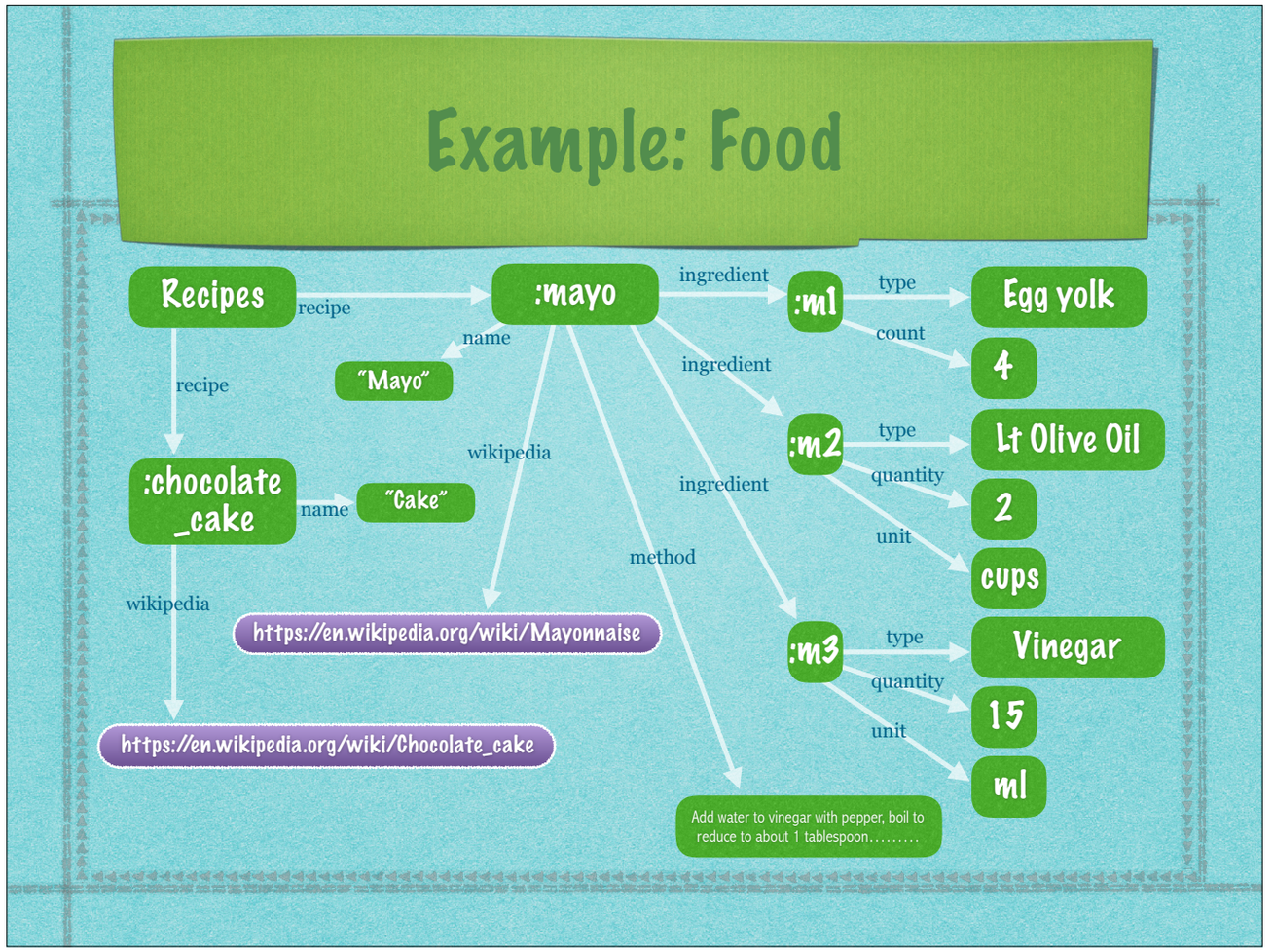
That first point means that they also allow ad hoc connections, which makes them very flexible as models evolve over time, and this is part of the reason why they are so good with complex relationships.



So let’s see an example. This is a small portion of a family. Families can have many different links representing redundant information, like nieces and uncles, so this isn’t complete, but it contains everything that we need.

You can see that the information is represented by a set of entities or nodes, which we label, and then links between nodes, which we also label. That’s it. The fact that the structure is so simple allows it to represent any kind of data at all.

Most of the nodes here represent people, but we can also add in other types of nodes, along with additional information that wasn’t considered in the initial graph. * For instance, we can add in some information that Cristy has a partner who is not a member of family, and together they have a company.



For something a little more complex, we can represent recipes. In a recipe, ingredients appear as a tuple, which includes measurement units and quantity. Each of these tuples is represented by an anonymous node. Some systems make it impossible to refer to these nodes directly * but most provide some kind of internal label for them.

* Using one of the ideas from the semantic web, data can also be linked to URLs, which lets us refer to external data as part of the graph. This is particularly interesting when those URLs form nodes in foreign graph databases, like Freebase, and this sort of reference is the foundation of Linked Data in the Semantic Web.

Query

- All recipes using ≤ 2 cups of flour

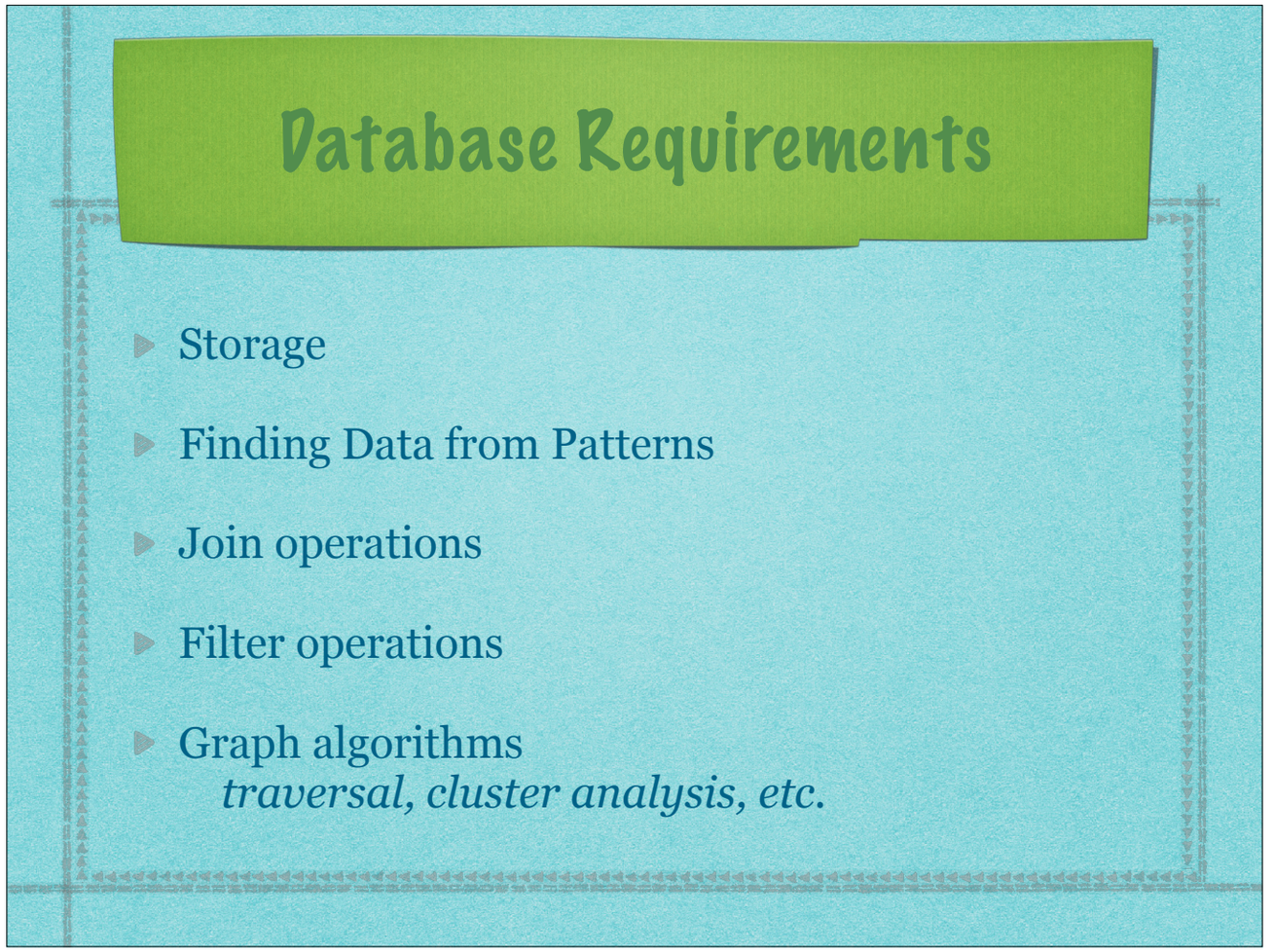
```
PREFIX recipe: <http:/example.com/food/recipe/>
PREFIX food: <http://example.com/food/types/>
WHERE [?recipe :ingredient ?i]
      [?i :type :flour]
      [?i :unit :cups]
      [?i :quantity ?q]
      [(<= ?q 2)]
SELECT ?recipe
```

```
PREFIX recipe: <http:/example.com/food/recipe/>
PREFIX food: <http://example.com/food/types/>
WHERE {?recipe recipe:ingredient ?i .
  ?i recipe:type food:flour .
  ?i recipe:unit recipe:cups .
  ?i recipe:quantity ?q .
  FILTER (?q <= 2) }
```

Now let’s have a look at an example query.

This is a Datomic query, which is also the syntax used by Datascript and Asami. Similarly to an SQL query, the WHERE clause of the query specifies most of the work. The major difference is that SQL automatically provides variables bound according to column names from a table, whereas graph queries use a template based on a triple of Entity-Attribute-Value to assign variables to columns, restricting the bindings to the rows that match the template. Filters then get applied to remove any sets of bindings that don’t match a given predicate, in this case, the binding for ?q being less than or equal to 2. At the end of the query, the variables of interest are the only ones that are kept, and these are declared in the FIND clause.

* By way of comparison, a SPARQL query has the same structure as a Datomic query, though with different syntax. It looks a little more like SQL with SELECT and WHERE, and the FILTER clause being explicitly labeled. Because XML style namespacing is used a lot, prefix declarations can make a query much more readable.



After seeing a little of how graph databases are used, we can consider what they need to provide.

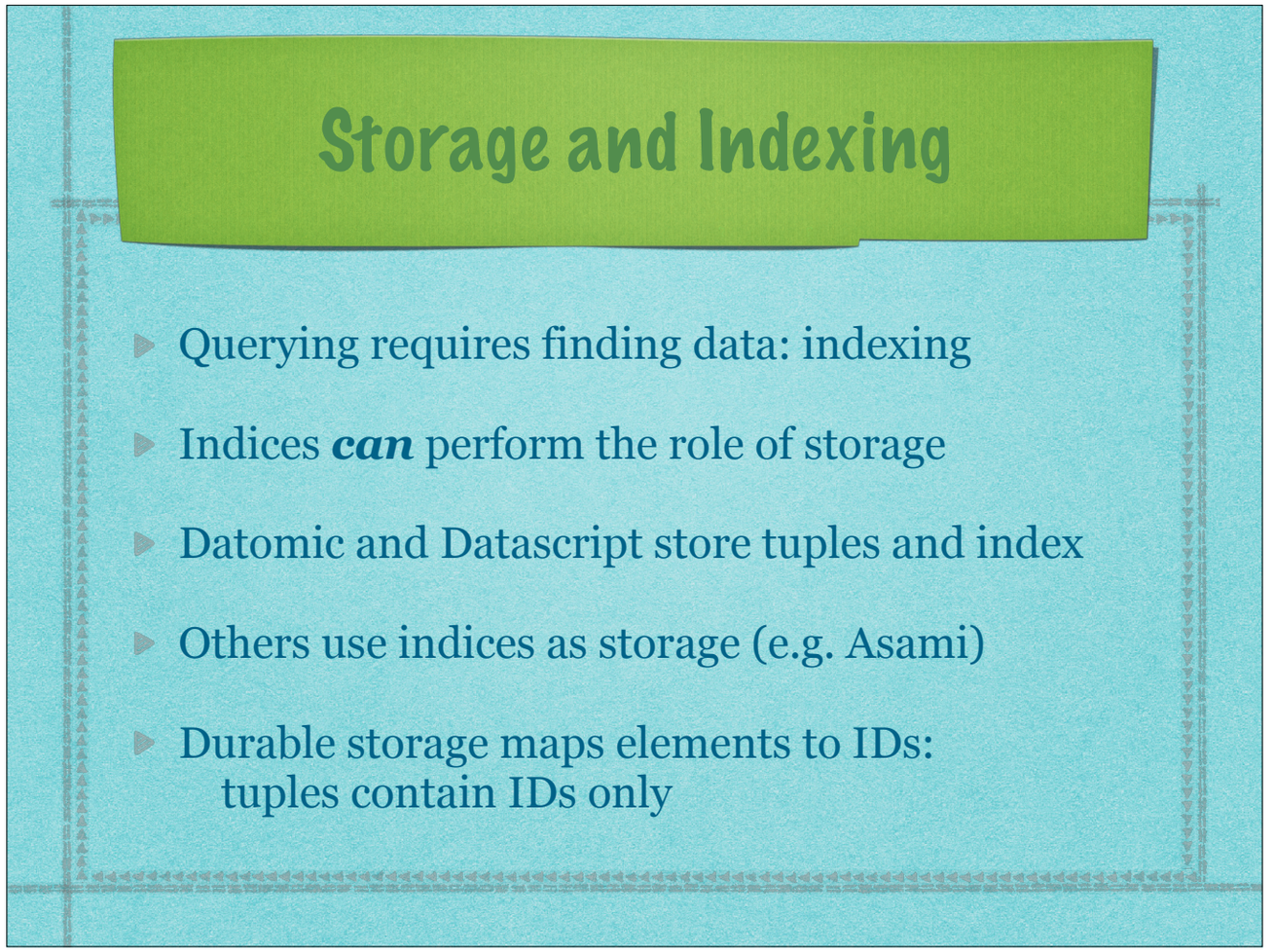
First off, they need to store the data of the graph. If it's to be durable, they will need on disk data structures to do that. We also saw in queries that they need to find a portion of that data that matches a template pattern. Queries also require a join operation where bindings for a variable can be matched. They need filter operations on results. And an increasingly common set of features includes operations for graph analysis, such as traversal and cluster identification.

Storage

- Graph represented as node IDs and edge ID
- A Triple
  Entity-Attribute-Value
  Subject-Predicate-Object
  Source-Property-Target
- Often stored as a 4-tuple (quad), 5-tuple, etc.
  Includes: graph labels, timestamps, metadata...

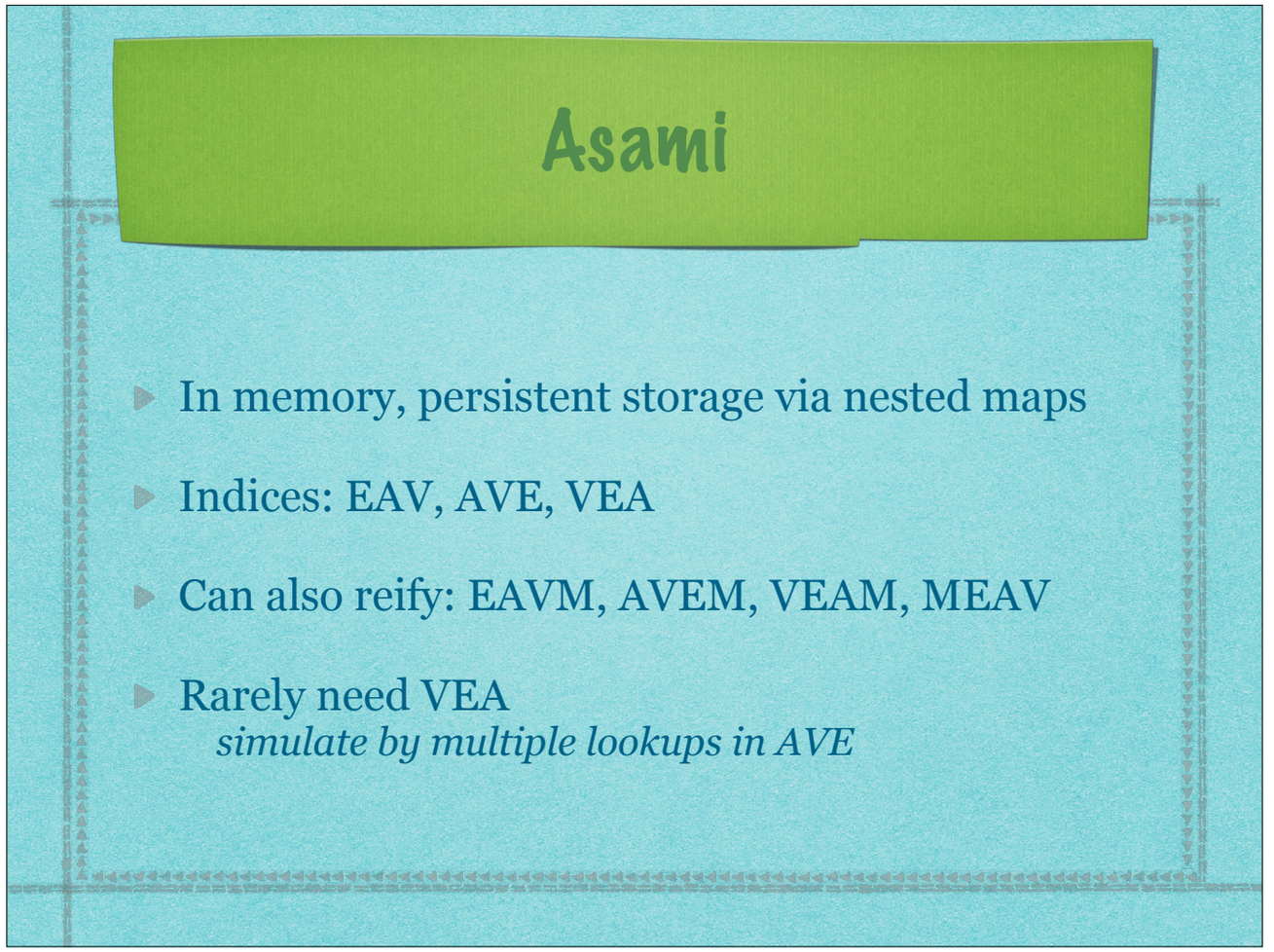
Graphs are stored in units of a single edge at a time. The node at the start and end of an edge are stored, along with the label of the edge. These form a triple. Depending on the system, these may be referred to as an Entity-Attribute-Value, like in Datomic, or a Subject-Predicate-Object, as in RDF, but it all means the same thing.

In most systems, these edges are represented along with some metadata, which may or may not be indexed. This can include identifiers for different graphs, timestamps for when the edge was created, and so on.



Getting a little more specific about storage, the best way to meet the needs of matching patterns in a query is to index the triples. Given the simplicity of triples, indices can often serve the purpose of also being the storage, though Datomic and Datascript don't it that way.

While I'm going to stick with Asami and Datascript, one thing that durable stores will usually do is to create an index that maps strings and other datatypes to a number, and then map that number back to the original data. It's those internal numbers are then the only thing that gets stored in a triple.



Now we can look at how Asami manages this.

Asami is currently in memory only, and it uses the common Clojure practice of nested maps. At the bottom layer the standard indices uses sets.

There are 6 possible index patterns for a triple, though only 3 are required. We use the natural 3 which store in the patterns of Entity first, Attribute second, and Value last, and also Attribute-Value-Entity and Value-Entity-Attribute. Using these 3 indices, any pattern can be identified. When reifying with metadata, we usually put this at the end, and optionally, add an extra index for finding patterns with given metadata. But I'm going to focus on the main indices without metadata.

Also, the Value-Entity-Attribute index is rarely ever needed. Instead, we can usually simulate it by finding statements in other indices and iterating over what is needed. This saves space, and at almost no cost, because it's rarely ever needed.

Asami Index

- Lookups return sequences of bindings
- Bindings as an ordered sequence
- Var names in metadata

```
(with-meta
  [[“Peter” 12]
   [“Mary” 11]
   [“Anne” 13]]
  {:vars [?name ?age]})
```

Patterns in a query result in lookups on indices. These lookups return a sequence of the things that the variables matched on, as a sequence of bindings for that variable. Each binding is a vector containing the values for the variable, in order, and then metadata is added containing the variable names in the same order as the bindings.

This is an example of a binding, containing the variables ?name and ?age. The bindings are a sequence of 3 individual bindings, with the first value of each being the name, and the second being the age.

```clojure
(defn index-add
  [idx a b c]
  (update-in idx
             [a b]
             (fn [v] (if (seq v) (conj v c) #{c}))))



(defrecord GraphIndexed [eav ave vea]
  Graph
  (graph-add [this entity attribute value]
    (let [new-eav (index-add eav entity attribute value)]
      (if (identical? eav new-eav)
        this
        (assoc this :eav new-eav
               :ave (index-add ave attribute value entity)
               :vea (index-add vea value entity attribute)))))

  (resolve-triple [this entity attribute value]
    (get-from-index this entity attribute value)))
```

Here is a slice of code that accomplishes the index and lookups in Asami.

You can see the index-add function just uses update-in. The updating function creates a unitary set when nothing existed beforehand, or conj's onto an existing set when one already existed in that position. The 3 arguments are abstracted from entity, attribute and value, because the same code is used for all 3 indices.

The graph itself is a record that implements the graph protocol. You can see how it holds the 3 indices. Adding a statement first adds it to the eav index, and only if that index changed does it go ahead and insert into the other 2 indexes.

Asking the graph to resolve the triple calls a helper function that we'll look at next.

```
(defn simplify [g & ks] (map #(if (st/vartest? %) ? :v) ks))
(defmulti get-from-index simplify)


(defmethod get-from-index [ ?  ?  ?]
  [{idx :eav} e a v]
  (for [e (keys idx), a (keys (idx e)), v ((idx e) p)]
    [e a v]))


(defmethod get-from-index [:v  ? :v]
  [{idx :vea} e a v]
  (map vector (get-in idx [v e])))


(defmethod get-from-index [ ? :v  ?]
  [{idx :ave} e a v]
  (let [edx (idx a)] (for [v (keys edx), e (edx v)]
    [e v])))


(defmethod get-from-index [:v :v :v]
  [{idx :eav} e a v]
  (if (get-in idx [e a v]) [[]] []))
```

The first step is a function that simplifies the template pattern into a form of holding a variable indicated by the question mark or a value, like a string or keyword. This form is used to determine which of 8 functions to use from the get-from-index multi method. There's a bit of redundancy here, so let's just consider a representative sample.

The first says that we have variable for all 3 parts of the template. This means that we want every edge of the graph. That's just a for loop that goes through all statements, and returns each as a vector for the binding.

The next one binds an entity and value, and looks for the connecting attributes. If 2 elements in a pattern are bound and there is only 1 variable, then the bindings are just the content of the set found with the get-in. Each row of bindings has to be a vector, so we wrap them by mapping vector over them.

If only one value is specified and there are 2 variables, the just looking up the top level of the index finds the data, and a for loop can pull it out, wrapping each binding in a vector as it goes.

Finally, if everything was bound, then it's just testing to see if a specific edge exists. A sequence of just one empty binding indicates that it was found, otherwise an empty seq is returned. This form works with the set algebra defined for bindings.

Query

Recipes containing <= 2 cups of flour

```
:find ?name
:where [?recipe :name ?name]
       [?recipe :ingredient ?i]
       [?i :unit :cups]
       [?i :quantity ?q]
       [?i :type :flour]
       [(<= ?q 2)]
```

Let’s look at how this plays out.

This query asks to find the names of recipes, where the recipe has an ingredient that is flour measured in cups, where 2 or fewer cups are needed.

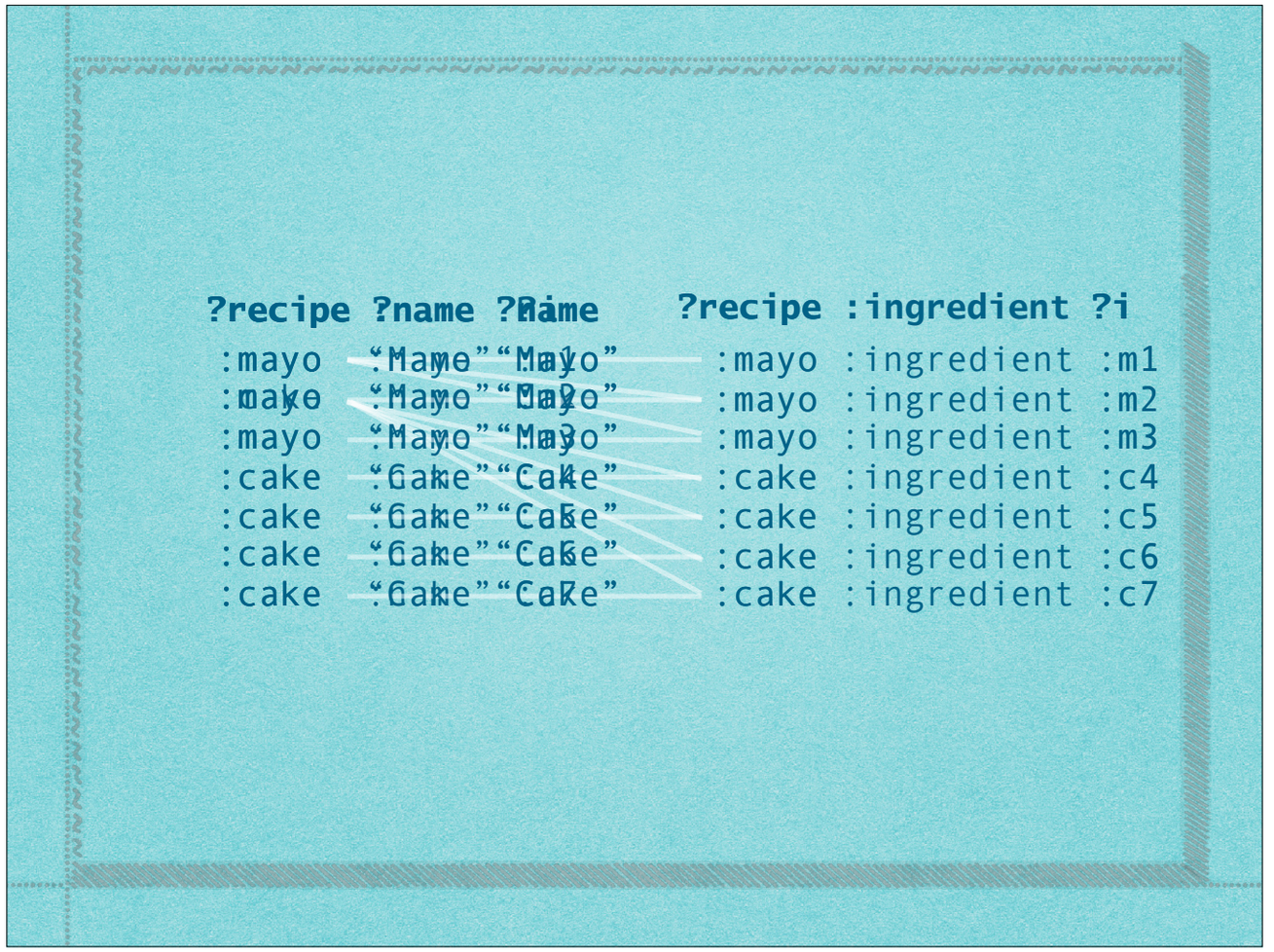
```
:mayo :ingredient :m1
:mayo :ingredient :m2
:mayo :ingredient :m3
:mayo :name “Mayo”
:cake :ingredient :c4
:cake :ingredient :c5
:cake :ingredient :c6
:cake :ingredient :c7
:cake :name “Cake”
:m1 :type :oil
:m1 :quantity 2
:m1 :unit :cups
:m2 :type :eggyolks
:m2 :count 4
:m3 :type :vinegar
:m3 :quantity 15
:m3 :unit :ml
:c4 :type :sugar
:c4 :quantity 1.5
:c4 :unit cups
:c5 :type :egg
:c5 :count 6
:c6 :type :flour
:c6 :quantity 1.5
:c6 :unit :cups
:c7 :type vanilla
:c7 :quantity 2
:c7 :unit :tsp
```

```
[?recipe :name ?name
[?recipe :ingredient ?i]
[?i :unit :cups]
[?i :quantity ?q]
[?i :type :flour]
[(<= ?q 2)]
```

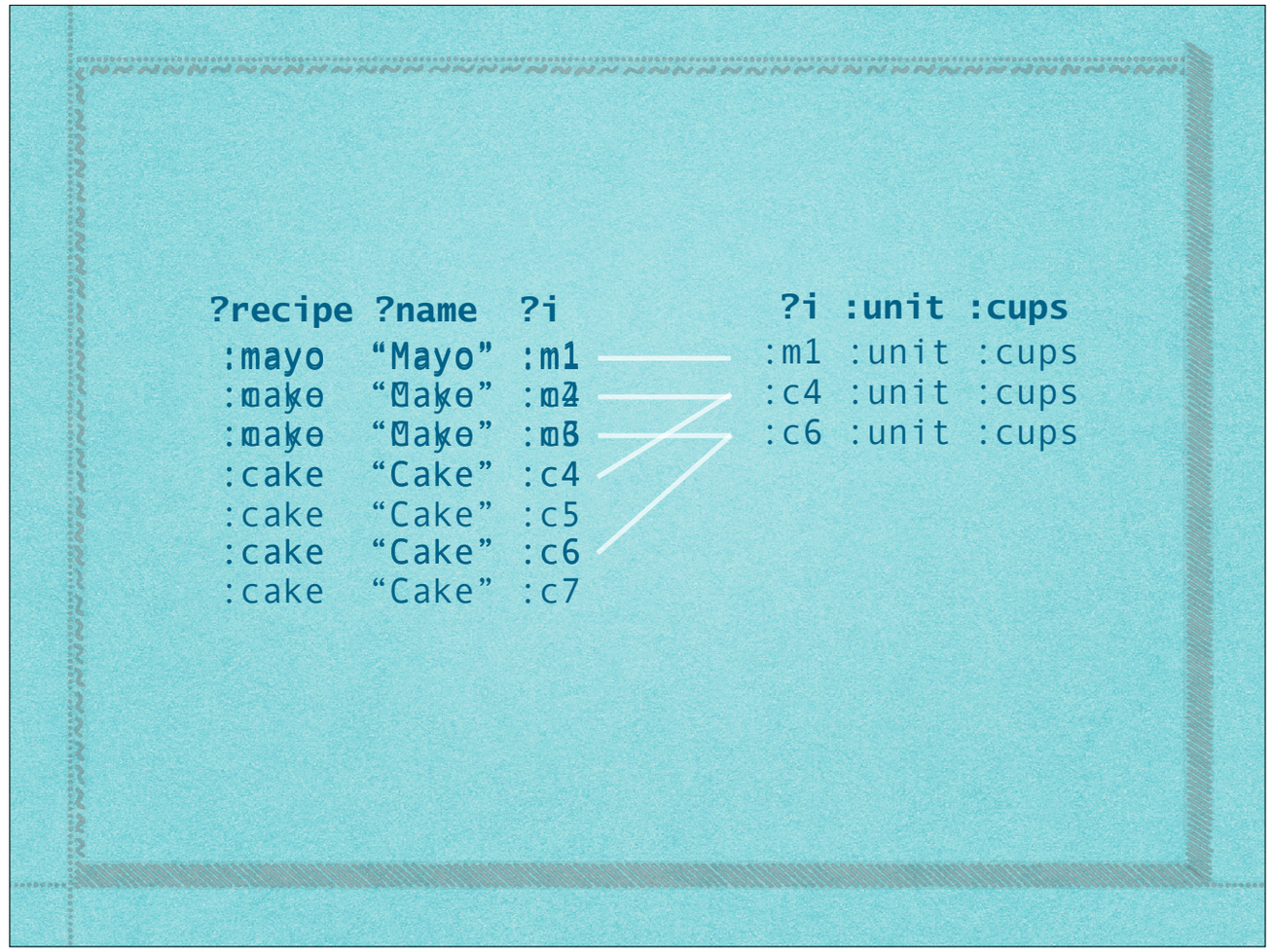
This shows all of the statements we have in our simple recipe graph.

* Each pattern in the WHERE clause matches part of the data
* First the names of recipes
* Then finding the ingredient nodes for those recipes
* Finding those nodes which use cups
* Getting the quantity
* And making sure they measure flour

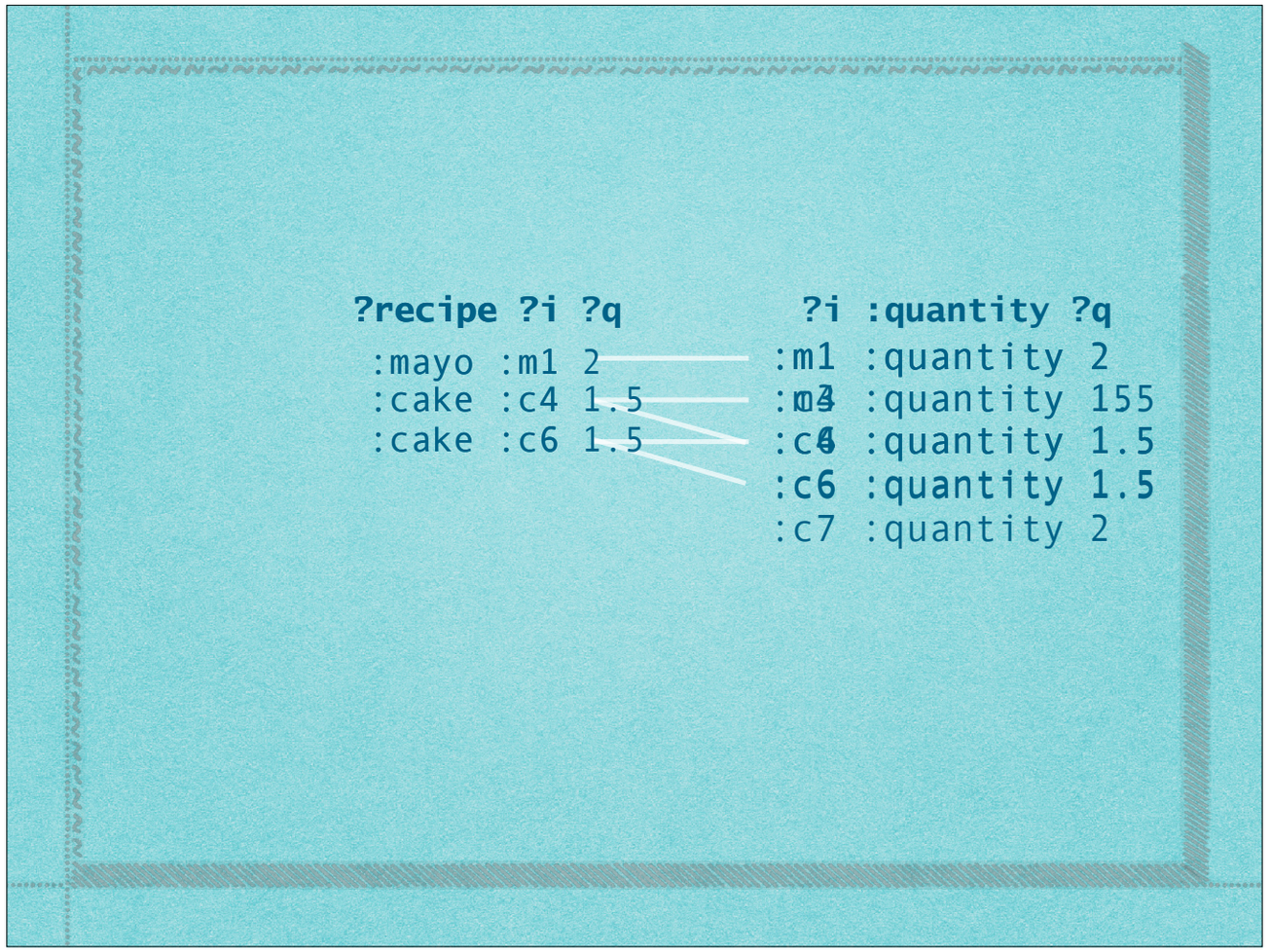
Filters will iterate over the result, just as any Clojure filter operation does.



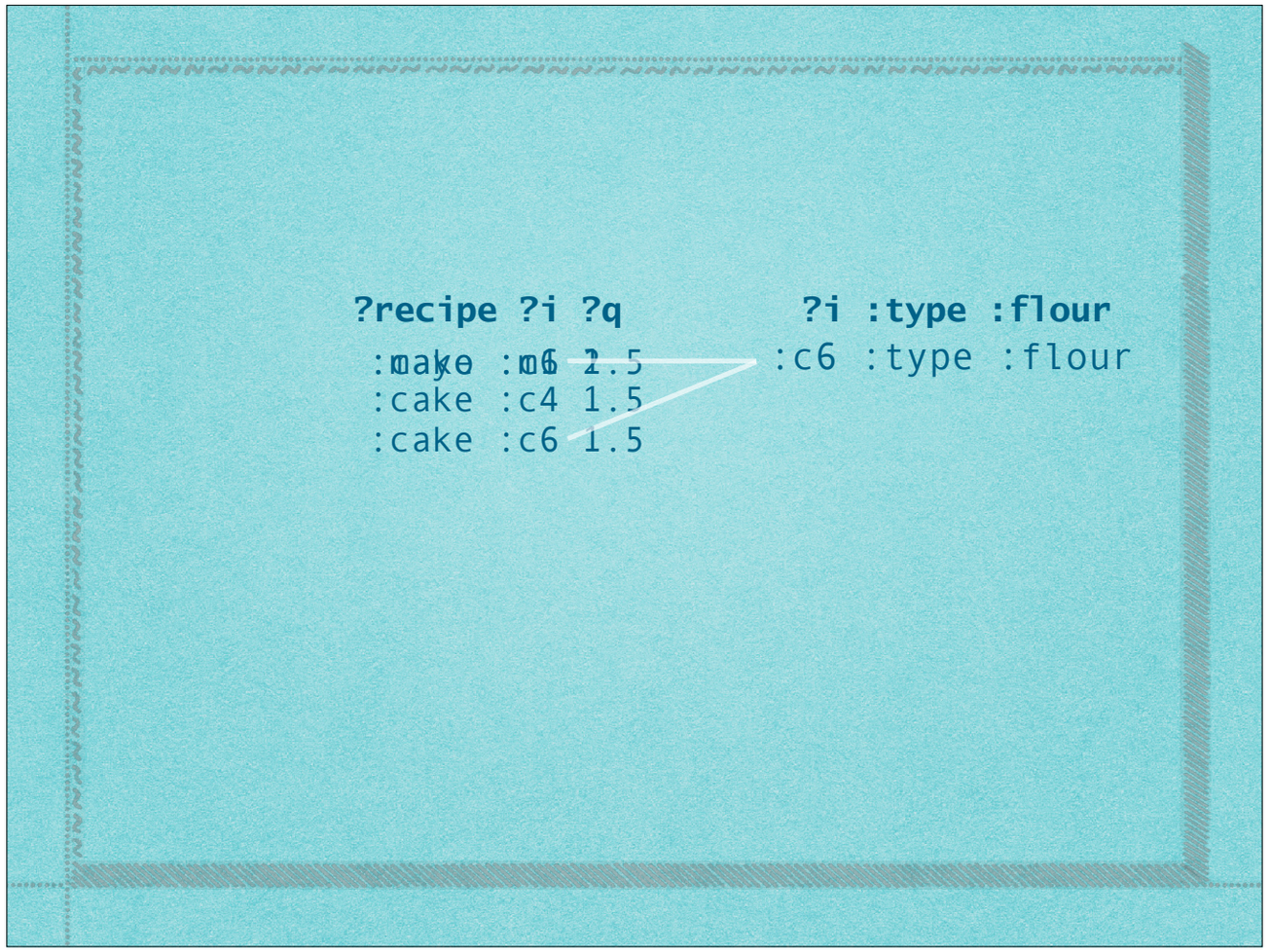
Now we can resolve the first 2 patterns and try to join them.
* first we find the matching recipe variable
* Then expand out the left hand side, repeating where needed so that everything on the other side is processed with its match.
* Then we remove the bound things, because they’re constant
* Remove the duplicate columns, and return that result



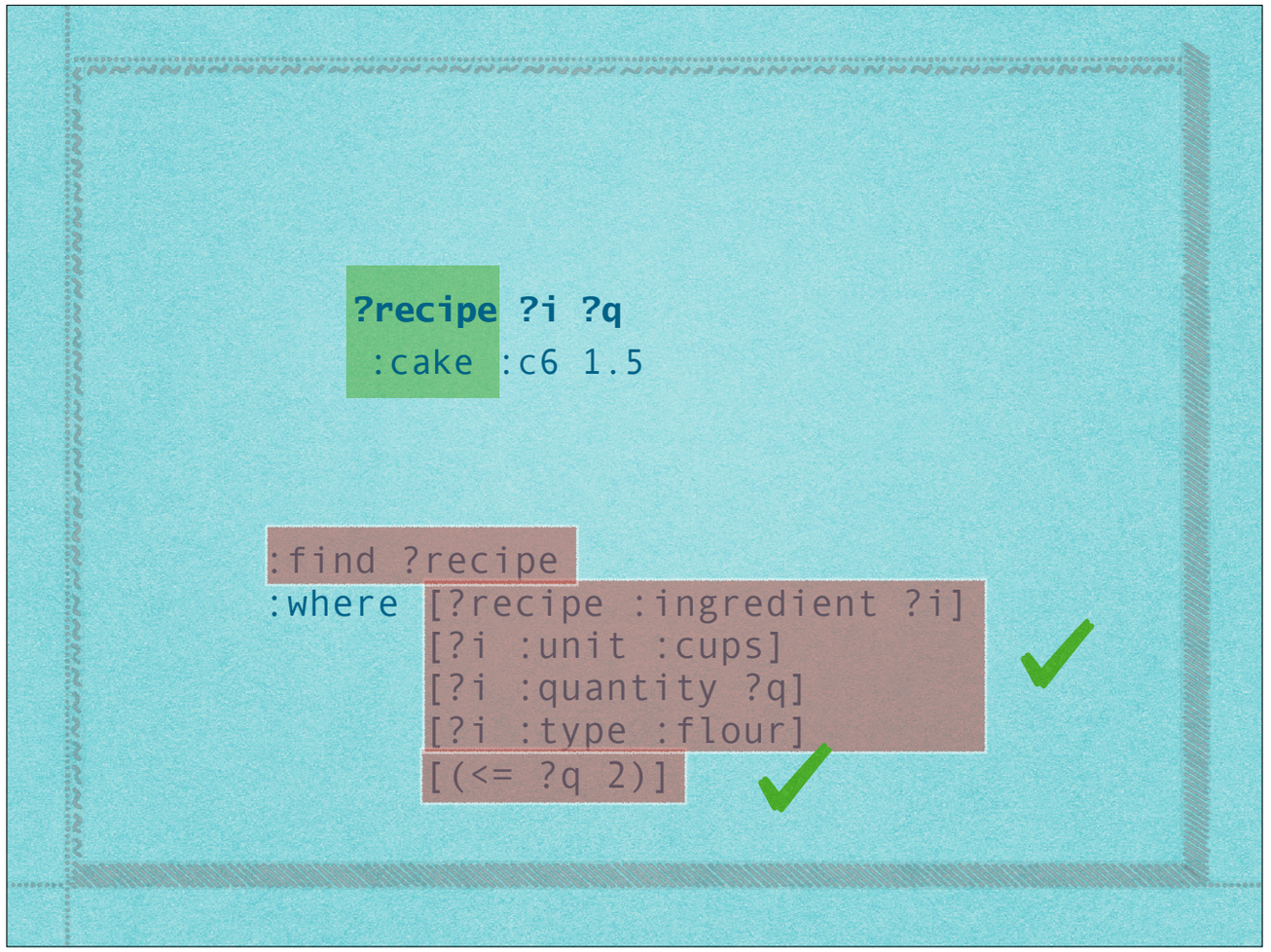
Now join this to the resolution of the next pattern.
* Find the matches for the bound variables.
* only 1 match on each side, so no repetition needed. Just remove the unneeded bindings
* remove the fixed data, and this is our new result



The next pattern finds the quantity
* Find the matching ingredient labels
* Remove the unnecessary bindings
* and remove the unneeded columns



Finally, it only wants the flour
* find the match
* remove the unneeded bindings
* and that’s the result of all the joins



Compare this back to the original query
* We looked at all the bindings
* And we joined them all
* You can see that the filter will pass the single value that was returned
* So that’s done
* Now we just need to handle the FIND clause. That’s just asking for the recipe column
* So we remove everything else and that’s done

That’s the entire query answered.

```
(defn pattern-left-join
  [graph partial-result pattern]
  (let [cols (:cols (meta partial-result))
        total-cols (calc-new-columns cols pattern)
        pattern->left (matching-vars pattern cols)]

    ;; iterate over partial-result, lookup pattern
    (with-meta
      (for [left-row partial-result
                  ;; convert bindings in left-row into the
                  ;; pattern to lookup in the graph
            :let [lookup (modify-pattern left-row
                                         pattern->left
                                         pattern)]
            right-row (gr/resolve-pattern graph lookup)]
        (concat left-row right-row))
      {:cols total-cols})))
```

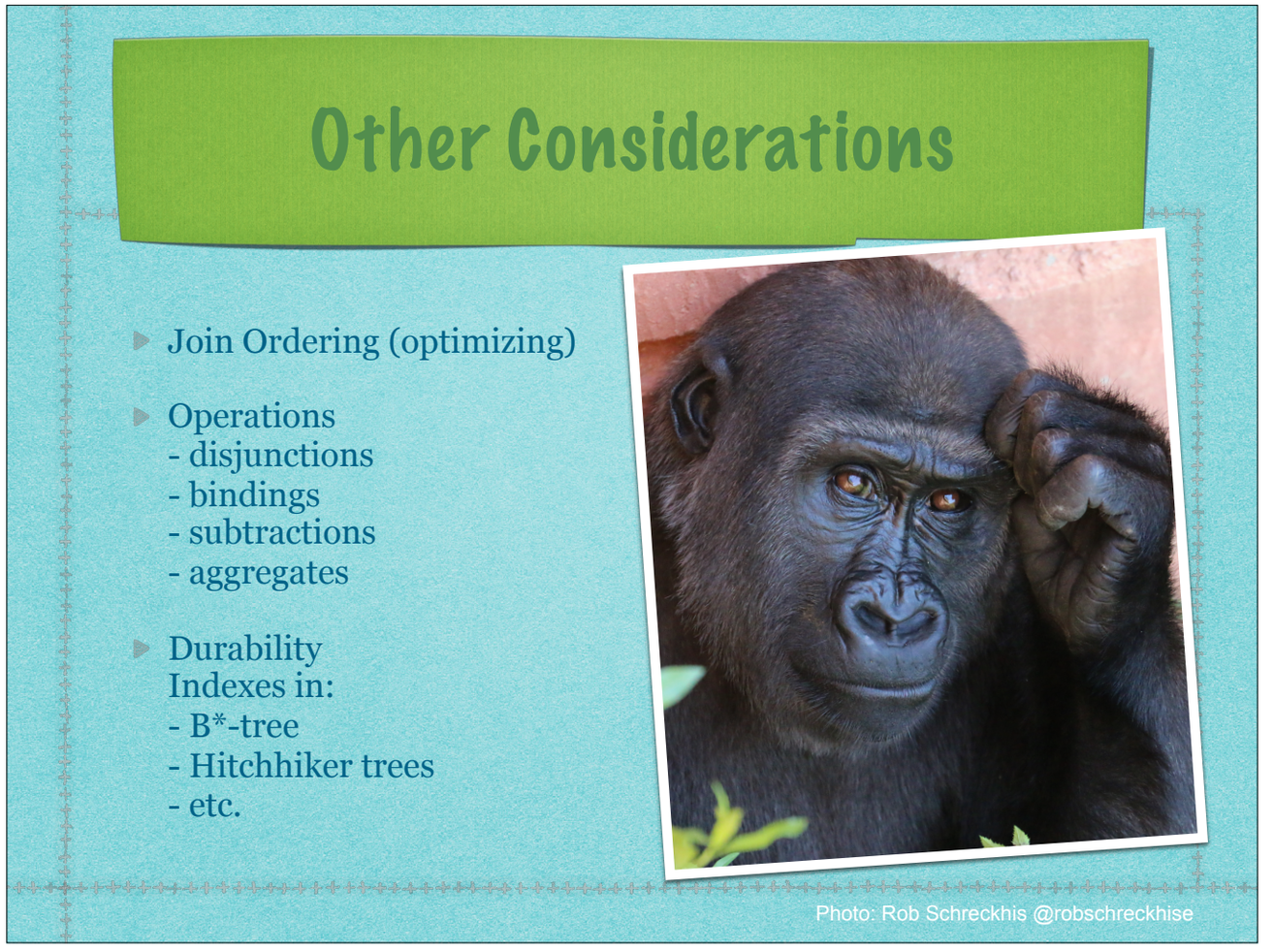
While negations, disjunctions, and other operations may all be considered important, the one operation that is fundamental to how graph databases work is the join.

This shows the basic code for that operation.
To start with the columns on the left are found, and the new columns on the right are appended to it. Then a mapping of offsets from the variables in the pattern on the right to the matching variables in the binding on the left is found. This is some simple numerical work, and isn't shown here.

The real work then happens by iterating over the left hand side. The right hand pattern isn't actually looked up in its raw form. It gets rewritten to use any of the variables that the left hand bindings already has. The new form is then looked up, and a concatenation of the left hand binding and each of the right hand results is returned.

This is a general enough approach, that if no variables matched, then it results in a cross product between patterns, which is actually what the math for this operation requires. That allows any ordering of patterns to still return the same results, even if it's a less efficient process.

Finally, the new columns labels are added as metadata



I just mentioned how inappropriate ordering can result in cross products, which gives a lot more data to iterate over when processing a query. This means that the order of operations in a join can be important. Datomic and Datascript leave this to the user, though query planning can be automated with varying degrees of success. Datomic and Datascript leave this to the user, though Asami does have a query planner by default.

Other operations are also important for these databases, including OR operations, binding values with functions, subtractions and aggregates. Asami has some of these, but not others. There is some complexity due to the query planner, and I am working on this at the moment. Most of them are features in the other graph databases.

Finally, an important consideration that I have not shown here is how these get built in durable indices. This is a complex topic, with lots of approaches, so I'm not going into it here. One thing I'd like to point out is that that Datomic has a very interesting abstraction of using key-value storage to create durable trees similar to those that it uses for in-memory persistent indexes.

Links

- Asami
  https://github.com/threatgrid/asami
- DataScript
  https://github.com/tonsky/datascript

Paula Gearon
@quoll https://github.com/quoll

What I just showed you is just the very basics to get a graph database going, but it should give you a little idea of what is going on inside the system. What I hope for is that you will be able to use this to decide if a graph database may meet your needs in the future and to understand what it's doing so that you can compare it to other systems.

I've provided a few links here, both for the Asami project that I was describing, as well as Datascript. If you look in there, you'll see some elements are different (for instance, they store datoms as records, and index them separately), but the ideas are very similar.