Video transcript Lesson 2.3 SPARQL in 11 minutes

Sparkle is a W3C standard. It stands for Sparkle Protocol and RDF Query Language. The protocol part is usually only an issue for people writing programs that pass Sparkle queries back and forth between different machines.

For most people, Sparkle's greatest value is as a query language for RDF, another W3C standard. RDF lets you describe data using a collection of three-part statements such as employee3 has a title of vice president. We call each statement a triple and we call its three parts the subject, predicate, and object.

You can think of them as an entity identifier, an attribute name, and an attribute value. The subject and predicate are actually represented using URIs to make it absolutely clear what we're talking about. URIs are kind of like URLs and often look like them, but they're not locators or addresses, they're just identifiers.

These URIs show that we mean employee3 from a specific company and that we mean title in the sense of job title and not a label for a book, movie, or other creative work because we're using the URI for title defined by the W3C's published version of the vCard business card ontology. The object, or third part of a triple, can also be a URI if you like. This way the same resource can be the object of some triples and the subject of others which lets you connect up triples into networks of data called graphs.

To make URIs simpler to write, RDF's popular turtle syntax often shortens the URIs by having an abbreviated prefix stand in for everything in the URI before the last part. Just about any data can be represented as a collection of triples. For example, we can usually represent each entry of a table by using the row identifier as the subject, the column name as the predicate, and the value as the object.

This can give us triples for every fact on the table. Some of the property names here come from the vCard vocabulary. For the properties not available in vCard, I made up my own property names using my own domain name.

RDF makes it easy to mix and match standard vocabularies and customizations. Let's say that of the employees on this table, John Smith completed the employee orientation course twice and we want to store both of his completed orientation values. This is no problem with RDF.

If the original data had been stored in a relational database table, adding John's second completed orientation value would have been a lot more difficult. Let's look at a simple SPARQL query that retrieves some of this data. We want a list of all the employees whose last name is Smith.

Like the turtle RDF syntax, SPARQL lets you define prefixes so that you don't have to write out full URIs in your queries. For most SPARQL queries, it's best to look at the where clause first because that describes which triples we want to pull from the data set that we're querying. The where clause does this with one or more triple patterns which are like triples with variables as wildcards substituted into one, two, or all three of each triples parts.

In this query, the one triple pattern will match against triples whose predicate is the family name property from the vcard vocabulary, whose object is the string Smith, and whose subject is anything at all because this triple pattern has a variable that I named person in this position. The select clause indicates which variables values we want listed after the query executes. This query only has one variable, so that's the one we want to see.

When the query executes, it finds two triples that match the specified pattern. So the query's results, that is the values that got assigned to the person variable, will be the subjects from those two triples. Let's execute the query and find out.

Who are these Smiths? The mp1 and mp2 identifiers listed in our first query's results don't tell us much. Let's add a second triple pattern that matches on the given name of the employees who matched the first triple pattern and stores that value in a new given name variable. We'll also adjust the select clause to request the value of our new variable.

After the query processor finds each triple that matches the first triple pattern, it will remember the value it stored in the person variable when it looks for triples that match the second one. Of the two people who have a triple that matches the first triple pattern, they each have a triple for given name as well, so when we run the query we see their given name values in the result. Let's retrieve the given name, family name, and hire date of all the employees.

We can do this with a where clause that has three triple patterns, one for each piece of information that we want to retrieve. If we want to narrow down the results based on some condition, we can use a filter pattern. We want a list of employees who were hired before March 1st, so the filter pattern specifies that we only want HD values that are less than March 1st, 2015, expressed as an ISO 8601 date format.

And that's what we get. Let's remove the filter condition and list the employees and their completed orientation values instead of their hire date values. We see Heidi and John's orientation dates, but the other employees don't appear at all in the results.

Why not? Let's look more closely at the query's triple patterns. The query first looks for a triple with a given name value, and then a triple with the same subject as the subject that it found to match the first triple pattern, but with a family name value, and then another triple with the same subject and a completed orientation value. John and Heidi each have triples that match all the query's triple patterns, but Francis and Jane cannot match all three triple patterns.

Note how John actually had two triples that matched the query's third pattern, which is why the query had two rows of results for him, one for each completed orientation value. What if we want to list all of the employees and, if they have any, their completed orientation values? We can tell the query processor that matching on the third triple pattern is optional. This query asks for everyone with a given name and a family name, and, if they have a completed orientation value, it will show that too.

Next, let's say that Heidi is scheduling a new orientation meeting and wants to know who to invite. In other words, she wants to list all employees who do not have a completed orientation value. Her query asks for everyone's given and family names, but only if, for any

employee who matches those first two triple patterns, no triple exists that lists a completed orientation value for that employee.

We do this with the key words not exists. And there they are, Jane and Francis. So far, the only way we've seen to store a value in a variable is to include that variable in a triple pattern for the query processor to match against some part of a triple.

We can use the bind keyword to store whatever we like in a variable. This can be especially useful when the bind expression uses values from other variables and calls some of Sparkle's broad range of available functions to create a new value. In this query, the bind statement uses Sparkle's concat function to concatenate the given name value stored by the first triple pattern, a space, and the family name value stored by the second triple pattern.

It stores the result of this concatenation in a new variable called Running the query creates a new full name value for each employer. All the queries we've seen so far have been select queries, which, as with SQL, list values that you want returned as columns in a table of results. A Sparkle construct query uses the same kind of where clauses that a select query can use, but it can use the values stored in the variables to create new triples.

Now note how the triple pattern showing the triple to construct is inside of curly braces. These curly braces can enclose multiple triple patterns, which is a common practice when, for example, a construct query takes data conforming to one model and creates triples conforming to another. This is great for data integration projects.

Sparkle can do a lot more than what we've seen here. You can use data types and language tags, you can sort and aggregate query results, and you can add, delete, and update data. You can also retrieve JSON, XML, and delimited versions of query results from query processors, and you can send queries to remote, private, or public data collections, and there are quite a few of those out there.

The O'Reilly book Learning Sparkle shows you how to do all this and more. You'll learn about what Sparkle can add to your applications, and what kind of open-source and commercial Sparkle processors are available. You'll also learn about what Sparkle and inferencing can offer each other, and what issues to consider when you want your queries to be more efficient.

And you'll learn where Sparkle fits into discussions of the semantic web and linked data. Go to learningsparkle.com to order Learning Sparkle as a bound hardcopy book or in a variety of ebook formats from Amazon or directly from O'Reilly Publishing.