SPARQL Anything tutorial

1 Introduction

SPARQL Anything is an extension of SPARQL which enables users to either query non-RDF data, i.e. using the SELECT query form in SPARQL, or to create an RDF ontology from the data, i.e. using the CONSTRUCT query form. The object of this study is to investigate the user experience with SPARQL Anything. For the study, we use the CONSTRUCT form and three data formats: CSV, JSON and XML.

In Section 2, we provide an overview of SPARQL Anything. Section 3 then briefly describes those features of SPARQL which are used in the study; these only represent a small part of the SPARQL standard. Sections 4, 5 and 6 describe how SPARQL Anything works with CSV, JSON and XML. Please note that the study questions will not be concerned with details of, e.g. SPARQL syntax, but with the creation of the appropriate graph patterns.

This tutorial contains everything you need to know to undertake the study. Please read through the tutorial before undertaking the study. However, there is no need to memorize the tutorial. You will be free to refer to it at any time during the study.

2 Overview of SPARQL Anything

SPARQL Anything uses the SERVICE operator to enable queries to a variety of data formats. As an example, for a JSON file, the SERVICE operator takes the form:

```
SERVICE <x-sparql-anything:location=example.json>
```

There are a variety of options available to be used with the SERVICE operator. However, in this study we are concerned with the basic features of SPARQL Anything, and will use only a subset of the options[1]. Note that the file extension is used to specify the data format. In the study we use .csv; .json; and .xml. A SPARQL Anything query can have more than one SERVICE keyword, referring to different files. Moreover, these files may be of different data formats.

In our study, the basic structure of a query with SPARQL Anything will be:

Set of PREFIX statements
CONSTRUCT clause, where the user defines the required ontology
WHERE clause with SERVICE operator(s)

3 SPARQL features used in the study

We assume that study participants are familiar with the basic features of SPARQL[2]. We explain here four particular features used in our tutorial example and in the study.

[1] For more detail about SPARQL Anything, beyond what is required for this study, see https://sparql-anything.readthedocs.io/en/latest/

[2] This study only uses a basic subset of SPARQL; for full details of the standard, see https://www.w3.org/TR/sparql11-query/

3.1 Type conversion and the use of BIND

In our study, we use two functions for type conversion: IRI() converts from a string to an IRI; xsd:integer() converts to an integer format. In this study, string literals in RDF will be represented in quotes, e.g. *"AceCo"*. Integers will either be represented as a sequence of digits, e.g. 2420 or in the form *"2420"^^xsd:int*[3].

We frequently use these functions in conjunction with the BIND statement. For example, we sometimes need to create IRIs by using a prefix, followed by a character string input from a data file. Figure 4, discussed in Section 4, has two cases of this, e.g.

```
PREFIX ex: http://example.com/
…
BIND(IRI(CONCAT(STR(ex:),?companyName)) AS ?company)
```

Here, we have a character string bound to the variable ?companyName, and we wish to prefix this with 'ex:'. Working from inside of the expression outwards, we first use the STR function to create a character string from 'ex:', i.e. to expand the prefix. The CONCAT function then concatenates this with the string which is bound to ?companyName, and the IRI function converts the resultant string to an IRI. The resultant IRI, which will be bound to ?company, will be represented as, e.g.

<http://example.com/AceCo>

Another example arises because numeric strings, in CSV and XML data, are represented as character strings. We need to convert these to integer literals in RDF. This is also illustrated in Figure 4, where we have:

```
BIND(xsd:integer(?revenueString) AS ?revenue) .
```

Here ?revenueString represents a string. The function xsd:integer converts this to integer, and the result is bound to ?revenue. Conversion to integer would be necessary if arithmetic operations are to be executed, or if we wish to compare an integer represented as a string with an integer proper. Note that conversion to integer is not necessary with integers read from a JSON file. Conversion to integer in such cases has no effect.

3.2 FILTER keyword

The FILTER keyword is used to select a subset of the otherwise possible solutions. In our study we use FILTER where we are taking data from two different files, and we wish to match data from one file with data from another. This is illustrated in our example in Figure 4.

3.3 RDF containers

SPARQL Anything uses RDF containers, e.g. to represent arrays in JSON and the relation between an element and its children in XML. Specifically, it uses the predicates rdf:_1, rdf:_2 etc to indicate container elements. For example, the triple 'C rdf:_1 M' indicates that the node M is a member of the container C.

[3] There is an implementation difference between these two formats. However, this is not relevant to our study.

3.4 Square bracket notation

In our discussion of the intermediate ontology ("triplification") created by SPARQL Anything, we use square brackets to avoid the explicit representation of blank nodes, see Figures 5, 9 and 13. Table 1 illustrates this notation, with the equivalent expression using blank nodes on the right-hand side. In the table, p and p_i are IRIs or variables; o and o_i are IRIs, literals or variables. Lines 1 and 2 are equivalent, and lines 3 and 4 show increasingly more complex expressions. In particular, line 4 shows the use of square brackets in conjunction with the use of semicolon to indicate that the same node is the subject of multiple triples.

Table 1: square bracket notation and equivalent with blank nodes

1	[] p o .	_:b1 p o .
2	[p o] .	_:b1 p o .
3	[p_1 [p_2 o]] .	_:b1 p_1 _:b2 . _:b2 p_2 o .
4	[p_1 o_1 ; p_2 [p_3 o_2 ; p_4 o_3] ; p_5 o_4] .	_:b1 p_1 o_1 . _:b1 p_2 _:b2 . _:b2 p_3 o_2 . _:b2 p_4 o_3 . _:b1 p_5 o_4 .

We also use the square bracket notation, although in a more compact form, in the graph patterns in our SPARQL queries, see Figures 4, 8 and 12. This has the advantage of avoiding the creation of variables which are not used elsewhere. For clarification, in each of the figures we repeat the relevant parts of the query with alternative expressions using additional variables. In your responses to the study questions, please feel free to use either square brackets, or to create additional variables.

4 SPARQL Anything with CSV

To illustrate the use of SPARQL Anything with CSV data, we take a small example. We have two CSV files: one containing company data, the other containing industry data. The companies file, 'companies.csv', contains information about three companies: name of company; annual revenue in millions of pounds sterling; and a code to indicate the sector in which the company operates. The industries file contains information about the three industries: the industry name and the industry code. In each file, the first row contains a header specifying the nature of the data in each column. The two files are shown in Figures 1 and 2.

```
company,revenue,industryCode
AceCo,2420,fi
BuildCo,10010,co
CableCo,990,te
```

Figure 1: companies.csv

```
industry,code
telecoms,te
finance,fi
construction,co
```

Figure 2: industries.csv

We wish to create an ontology as shown in Figure 3. The predicate ex:hasRevenue is used to link companies to their revenues, represented as integers; this is entirely defined by the companies file. However, we wish also to link companies and their industries, and for this we need to equate 'industryCode' in the companies file with 'code' in the industries file.

```
ex:AceCo     ex :hasIndustry  ex:finance ;
             ex :hasRevenue   2420 .
ex :BuildCo  ex :hasIndustry  ex :construction ;
             ex :hasRevenue   10010 .
ex:CableCo   ex:hasIndustry  ex:telecoms ;
             ex:hasRevenue   990 .
```

Figure 3: the required ontology

```
PREFIX xyz: <http://sparql.xyz/facade-x/data/>
PREFIX ex: <http://example.com/>
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
        ?company ex:hasRevenue ?revenue .
        ?company ex:hasIndustry ?industry .
} WHERE {
 SERVICE <x-sparql-anything:csv.headers=true,location=./data/companies.csv> {
        []      xyz:company ?companyName;
                xyz:revenue ?revenueString;
                xyz:industryCode ?industryCode .
 }
 SERVICE <x-sparql-anything:csv.headers=true,location=./data/industries.csv> {
        []      xyz:industry ?industryName;
                xyz:code ?code .
 }
        BIND(IRI(CONCAT(STR(ex:),?companyName)) AS ?company) .
        BIND(IRI(CONCAT(STR(ex:),?industryName)) AS ?industry) .
        BIND(xsd:integer(?revenueString) AS ?revenue) .
        FILTER (?industryCode = ?code) .
}
```

Alternative SERVICE statements using additional variables

```
 SERVICE <x-sparql-anything:csv.headers=true,location=./data/companies.csv> {
        ?b1 xyz:company ?companyName .
        ?b1 xyz:revenue ?revenueString .
        ?b1 xyz:industryCode ?industryCode .
 }
 SERVICE <x-sparql-anything:csv.headers=true,location=./data/industries.csv> {
        ?b2 xyz:industry ?industryName .
        ?b2 xyz:code ?code .
 }
```

Figure 4: SPARQL Anything query for use with CSV files in Figures 1 and 2

Figure 4 shows the SPARQL Anything query to create Figure 3 from Figures 1 and 2. At the top we show this using the square bracket notation. At the bottom of the figure we repeat the SERVICE statements using additional variables ?b1 and ?b2. In particular, note:

1. The CONSTRUCT clause defines the two triple forms required in the ontology.
2. The arguments to the two SERVICE operators specify:
    a. that SPARQL Anything is to be used;
    b. that the CSV files have header rows;
    c. and the locations of the data files.
3. After each of the SERVICE keywords there is a graph pattern. These begin with an node, represented by [] in the top version, ?b1 and ?b2 in the alternative version. In the case of the first graph pattern, this node is linked by three predicates to three variables. The predicates are formed using the xyz: prefix (defined in the PREFIX list).
4. The three BIND statements are necessary because the contents of a CSV file are interpreted as character strings. However, the ontology in the CONSTRUCT clause contains two IRIs (?company and ?industry) and one integer (?revenue). It is therefore necessary to explicitly convert from string to IRI or integer, as explained in sub-section 3.1.
5. Finally, the FILTER statement specifies that, for the solutions to the CONSTRUCT query, the *industryCode* in companies.csv must equate to the *code* in industries.csv.

To understand the graph patterns in (3) above, it is necessary to understand that SPARQL Anything operates by triplifying the data, i.e. by automatically creating an ontology. This intermediate ontology is internal to SPARQL Anything, i.e. it is not output. However, it is available to be queried by the graph pattern after the SERVICE keyword.

```
@prefix xyz:    <http://sparql.xyz/facade-x/data/> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

[ a fx:root ;
  rdf:_1 [ xyz:company “AceCo”;
           xyz:revenue “2420”;
           xyz:industryCode “fi”
         ];
  rdf:_2 [ xyz:company “BuildCo”;
           xyz:revenue “10010”;
           xyz:industryCode “co”
         ];
  rdf:_3 [ xyz:company “CableCo”;
           xyz:revenue “990”;
           xyz:industryCode “te”
         ];
].
```

first node in first graph pattern of Figure 4 binds to this node …

… and to this node …

… and to this node.

Figure 5: intermediate ontology created from the data of Figure 1

Figure 5 shows the intermediate ontology created by SPARQL Anything from companies.csv. In summary, the document is regarded as a collection, represented by the root node of the

intermediate ontology, with elements of the collection corresponding to the rows of the data. The first triple indicates that this node is of type fx:root. This root node is then linked by the predicates rdf:_1, rdf:_2 etc, to a node representing each row of the data. These nodes are, in turn, linked by predicates xyz:company, xyz:revenue, and xyz:industry to strings representing the company name, revenue, and industry. Note that the subject of the graph pattern after the first SERVICE keyword (i.e. [] or ?b1) matches the nodes representing the rows of the data file, i.e. the subjects of the triples with predicates xyz:company, xyz:revenue, and xyz:industry. Thus, we have three sets of solutions, one for each row of the file. Similar comments apply to the ontology created by SPARQL Anything from industries.csv, and to the graph pattern after the second SERVICE keyword.

5 SPARQL Anything with JSON

To describe how SPARQL Anything works with JSON, we use the same company and industry information as in the previous section, represented as the JSON files in Figures 6 and 7. The intention is to produce the same final ontology as in the previous example, i.e. as shown in Figure 3. The SPARQL query is shown in Figure 8.

```
{
        "companies": [
                {
                        "company": "AceCo",
                        "revenue": 2420,
                        "industryCode": "fi"
                },
                {
                        "company": "BuildCo",
                        "revenue": 10010,
                        "industryCode": "co"
                },
                {
                        "company": "CableCo",
                        "revenue": 990,
                        "industryCode": "te"
                }
        ]
}
```

Figure 6: companies.json

```
{
        "industries": [
                {
                        "industry": "telecoms",
                        "code": "te"
                },
                {
                        "industry": "finance",
                        "code": "fi"
                },
                {
                        "industry": "construction",
                        "code": "co"
                }
        ]
}
```

Figure 7: industries.json

The query shown in Figure 8 is very similar to the query shown in Figure 5 for the CSV file. The lines with the SERVICE keywords are amended to indicate the use of a JSON file. Additionally, there is no requirement to convert the revenue to integer format, because numeric strings in JSON, when not enclosed in quotes, are treated as numbers.

The structure of the intermediate ontology differs from that created from the CSV file. Figure 9 shows a root node representing the document. This root node is of type fx:root and is also the subject of a triple with predicate xyz:companies. The object of this triple is an RDF collection representing the JSON 'companies' array. In turn, this node is the subject of three triples with predicates rdf:_1, rdf:_2 and rdf:_3. In each case the object of these triples is a node representing an element of the companies array. These nodes are subjects of triples with

```
PREFIX xyz: <http://sparql.xyz/facade-x/data/>
PREFIX ex: <http://example.com/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
        ?company ex:hasRevenue ?revenue .
        ?company ex:hasIndustry ?industry .
} WHERE {
 SERVICE <x-sparql-anything:location=./data/companies.json> {
        []      xyz:company ?companyName;
                xyz:revenue ?revenue;
                xyz:industryCode ?industryCode .
 }
 SERVICE <x-sparql-anything:location=./data/industries.json> {
        []      xyz:industry ?industryName;
                xyz:code ?code .
 }
        BIND(IRI(CONCAT(STR(ex:),?companyName)) AS ?company) .
        BIND(IRI(CONCAT(STR(ex:),?industryName)) AS ?industry) .
        FILTER (?industryCode = ?code) .
}
```

Alternative SERVICE statements using additional variables

```
 SERVICE <x-sparql-anything:location=./data/companies.json> {
        ?b1 xyz:company ?companyName .
        ?b1 xyz:revenue ?revenue .
        ?b1 xyz:industryCode ?industryCode .
 }
 SERVICE <x-sparql-anything:location=./data/industries.json> {
        ?b2 xyz:industry ?industryName .
        ?b2 xyz:code ?code .
 }
```

Figure 8: SPARQL Anything query for use with JSON files in Figures 6 and 7

predicates xyz:company, xyz:revenue, and xyz:industryCode. The result is that, whereas in Figure 5 the CSV file is represented as a collection, in Figure 9 it is the companies array, within the file, which is represented as a collection. There is thus an extra level of nesting in Figure 9, compared to Figure 5.

Returning to the SPARQL query of Figure 8, the subject of the graph pattern, i.e. the pattern node represented by [] or ?b1, matches the nodes representing the elements of the companies array, i.e. the nodes which are the objects of triples with predicates rdf:_1, rdf:_2, and rdf:_3. This creates the required three sets of bindings to the variables ?companyName, ?revenue, and ?industryCode.

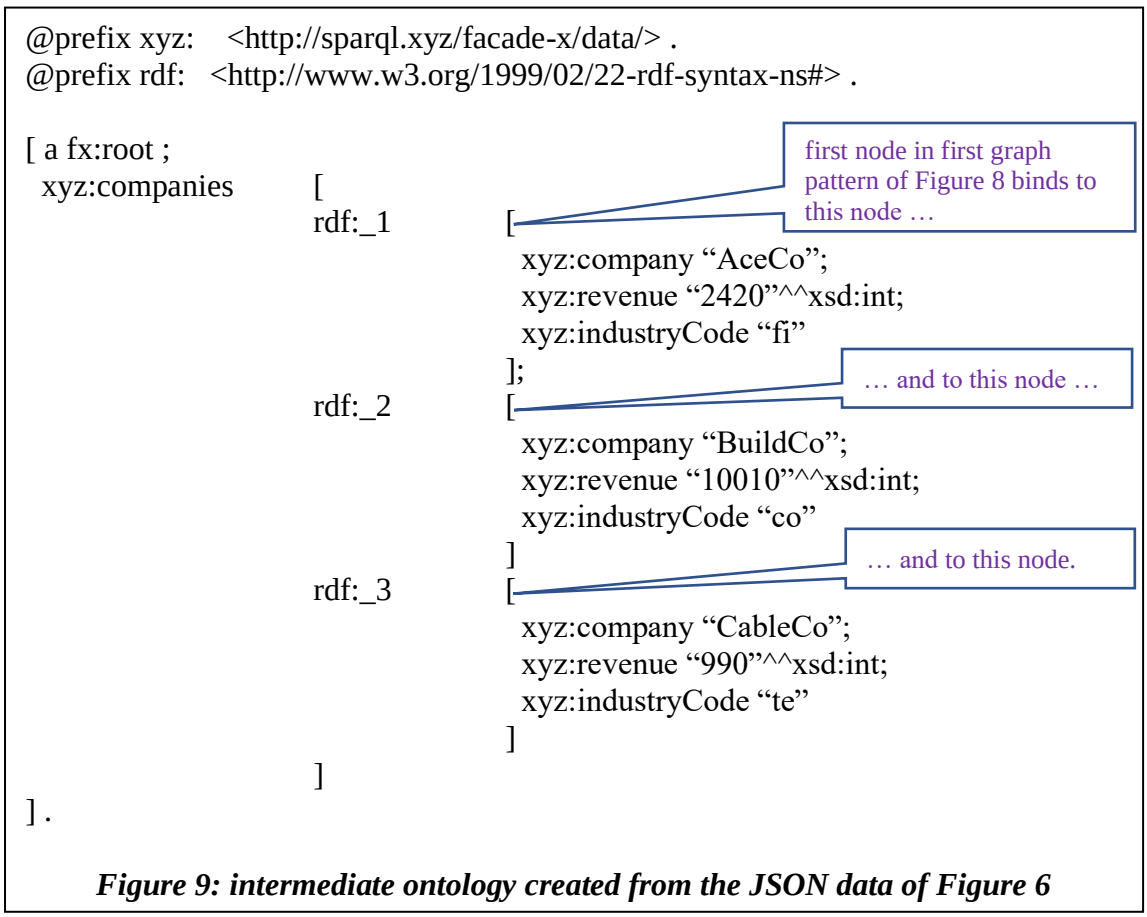

Figure 9: intermediate ontology created from the JSON data of Figure 6

6 SPARQL Anything with XML

In subsection 6.1, we illustrate the use of SPARQL Anything with XML files which contain tags, but no attributes. In subsection 6.2, we illustrate the use of SPARQL Anything with XML files which contain both tags and attributes.

6.1 XML with tags only

We use the same company and industry information as in the previous two examples, represented by the XML files in Figures 10 and 11, and the intention is to create the same final ontology, as shown in Figure 3.

```
<companies>
        <item>
                <company>AceCo</company>
                <revenue>2420</revenue>
                <industryCode>fi</industryCode>
        </item>
        <item>
                <company>BuildCo</company>
                <revenue>10010</revenue>
                <industryCode>co</industryCode>
        </item>
        <item>
                <company>CableCo</company>
                <revenue>990</revenue>
                <industryCode>te</industryCode>
        </item>
</companies>
```

Figure 10: companies.xml

```
<industries>
        <item>
                <industry>telecoms</industry>
                <code>te</code>
        </item>
        <item>
                <industry>finance</industry>
                <code>fi</code>
        </item>
        <item>
                <industry>construction</industry>
                <code>co</code>
        </item>
</industries>
```

Figure 11: industries.xml

Figure 12 shows the SPARQL Anything query. Like the CSV example, it requires a BIND statement to convert from XML character string data to integer. More significantly, the query differs from the two previous queries, in that the graph patterns in the SERVICE statements are more complex. We can see the reason for this by considering the intermediate ontology generated by SPARQL Anything from companies.xml, and shown in Figure 13.

```
PREFIX xyz: <http://sparql.xyz/facade-x/data/>
PREFIX ex:  <http://example.com/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

CONSTRUCT {
        ?company ex:hasRevenue ?revenue .
        ?company ex:hasIndustry ?industry .
} WHERE {
 SERVICE <x-sparql-anything:location=./data/companies.xml> {
        []      ?li1 [a xyz:company; rdf:_1 ?companyName];
                ?li2 [a xyz:revenue; rdf:_1 ?revenueString];
                ?li3 [a xyz:industryCode; rdf:_1 ?industryCode] .
 }
 SERVICE <x-sparql-anything:location=./data/industries.xml> {
        []      ?li4 [a xyz:industry; rdf:_1 ?industryName];
                ?li5 [a xyz:code; rdf:_1 ?code] .
 }
        BIND(IRI(CONCAT(STR(ex:),?companyName)) AS ?company) .
        BIND(IRI(CONCAT(STR(ex:),?industryName)) AS ?industry) .
        BIND(xsd:integer(?revenueString) AS ?revenue) .
        FILTER (?industryCode = ?code) .
}
```

Alternative SERVICE statements using additional variables

```
 SERVICE <x-sparql-anything:location=./data/companies.xml> {
        ?b1 ?li1 ?b2 . ?b2 a xyz:company . ?b2 rdf:_1 ?companyName .
        ?b1 ?li2 ?b3 . ?b3 a xyz:revenue . ?b3 rdf:_1 ?revenueString .
        ?b1 ?li3 ?b4 . ?b4 a xyz:industryCode . ?b4 rdf:_1 ?industryCode .
 }
 SERVICE <x-sparql-anything:location=./data/industries.xml> {
        ?b5 ?li4 ?b6 . ?b6 a xyz:industry . ?b6 rdf:_1 ?industryName .
        ?b5 ?li5 ?b7 . ?b7 a xyz:code . ?b7 rdf:_1 ?code .
 }
```

Figure 12: SPARQL Anything query for use with XML files in Figures 10 and 11

In the case of XML the company, revenue and industry tags are not used to generate predicate IRIs, as for CSV and JSON. Instead, they are used to generate IRIs representing classes, which form the objects of triples with predicate rdf:type, here shortened to ‘a’.

In summary, Figure 13 shows a root node. This is identified as being of type fx:root, but also of type xyz:companies. This node represents a collection; the three triples, with predicates rdf:_1, rdf:_2, and rdf:_3, have as objects the elements of this collection. These elements are themselves collections of type xyz:item;. Each of these collections has three elements, indicated again by a further set of triples with predicates rdf:_1, rdf:_2, and rdf:_3. These elements are collections of types xyz:company; xyz:revenue and xyz:industryCode. Each of these collections has only one element, indicated by the predicate rdf:_1; the object of these predicates are the data values.

Our first graph pattern in Figure 12 has a node, represented by [] or ?b1, which needs to bind to a blank node in Figure 13 which is connected by some predicate to a container of type xyz:company. This identifies the three nodes, as shown in Figure 13. The first, and in fact only, element of each container of type xyz:company is then bound to ?companyName. The three nodes are also connected by predicates to containers of type xyz:revenue and xyz:industryCode, each with one element which binds to ?revenueString and ?industryCode respectively.

Looking again at Figures 12 and 13, we see that we could have replaced ?l1, ?l2, ?l3 with rdf:_1, rdf:_2, and rdf:_3 respectively; we chose to be avoid being specific about the nature of these predicates.

Conversely, we could have replaced each of the occurrences of rdf:_1 with additional variables. However, this would cause ?companyName, ?revenueString and ?industryCode to be bound also to xyz:company, xyz:revenue and xyz:industryCode respectively, with the additional variables bound to the instances of rdf:type. We would then need FILTER statements to remove the unrequired bindings.

Another possible variant would be:

```
[]      a xyz:item; ?li1 [a xyz:company; rdf:_1 ?companyName];
                    ?li2 [a xyz:revenue; rdf:_1 ?revenueString];
                    ?li3 [a xyz:industryCode; rdf:_1 ?industryCode] .
```

This makes it clear that the initial node in the graph pattern is of type xyz:item. However, given the structure of the data, we do not need to specify this.

```
@prefix xyz: <http://sparql.xyz/facade-x/data/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xyz: <http://sparql.xyz/facade-x/data/> .
@prefix fx:  <http://sparql.xyz/facade-x/ns/> .

[ a xyz:companies, fx:root;
  rdf:_1 [ a xyz:item;
           rdf:_1 [ a xyz:company;
                    rdf:_1 “AceCo”
                  ];
           rdf:_2 [ a xyz:revenue;
                    rdf:_1 “2420”
                  ];
           rdf:_3 [ a xyz:industryCode;
                    rdf:_1 “fi”
                  ];
  rdf:_2 [ a xyz:item;
           rdf:_1 [ a xyz:company;
                    rdf:_1 “BuildCo”
                  ]:
           rdf:_2 [ a xyz:revenue;
                    rdf:_1 “10010”
                  ];
           rdf:_3 [ a xyz:industryCode;
                    rdf:_1 “co”
                  ];
         ];
 rdf:_3 [ a xyz:item;
           rdf:_1[ a xyz:company;
                    rdf:_1 “CableCo”
                  ];
           rdf:_2[ a xyz:revenue;
                    rdf:_1 “990”
                  ];
           rdf:_3[ a xyz:industryCode;
                    rdf:_1 “te”
                  ];
         ];
] .
```

first node in first graph pattern of Figure 12 binds to this node …

… and to this node …

… and to this node.

Figure 13: intermediate ontology created from the XML data of Figure 10

6.2 XML with tags and attributes

Once again, we use the same company and industry information as in the previous examples. This time we use a combination of XML tags and attributes, as shown for the revised companies.xml and industries.xml files in Figures 14 and 15. The intention is to create the same final ontology, as shown in Figure 3.

```
<companies>
        <company name = “AceCo” revenue = “2420” industryCode = “fi” />
        <company name = “BuildCo” revenue = “10010” industryCode = “co” />
        <company name = “CableCo” revenue = “990” industryCode = “te” />
</companies>
```

Figure 14: companies.xml

```
<industries>
        <industry name = “telecoms” code = “te” />
        <industry name = “finance” code = “fi” />
        <industry name = “construction” code = “co” />
</industries>
```

Figure 15: industries.xml

Figure 16 shows the SPARQL Anything query. Like the CSV and previous XML examples, it requires a BIND statement to convert from XML character string data to integer. However, this time, the query patterns after the SERVICE statements are simpler. We can see the reason for this by considering the intermediate ontology generated by SPARQL Anything from companies.xml, as shown in Figure 17. As already explained, the tags (in this case companies and company) are used to generate IRIs representing classes, which form the objects of triples with predicates rdf:type, again shortened to ‘a’. Additionally, the attributes are used to generate predicates, similarly to the use of JSON names to generate predicates.

The intermediate ontology contains a collection, of type xyz:companies, with three components. Each of these components is represented by a node of type xyz:company. These nodes then have predicates xyz:name, xyz:revenue, and xyz:industryCode. The SPARQL Anything query makes use of these predicates.

Because the data structures in the two files are relatively simple, the query only uses the predicates generated from the attributes. There is no use of the type information generated from the tags. However, in more complex situations it may be necessary to use both kinds of information. In Figure 16, after the first SERVICE statement, we could have written:

```
[]      a xyz:company;
        xyz:name ?companyName;
        xyz:revenue ?revenueString;
        xyz:industryCode ?industryCode .
```

However, in this simple case, the first line above is not necessary to uniquely identify the relevant nodes.

```
PREFIX xyz: <http://sparql.xyz/facade-x/data/>
PREFIX ex:  <http://example.com/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fx:     <http://sparql.xyz/facade-x/ns/>

CONSTRUCT {
        ?company ex:hasRevenue ?revenue .
        ?company ex:hasIndustry ?industry .
} WHERE {
 SERVICE <x-sparql-anything:location=./data/companies.xml> {
        []      xyz:name ?companyName;
                xyz:revenue ?revenueString;
                xyz:industryCode ?industryCode .
}
 SERVICE <x-sparql-anything:location=./data/industries.xml> {
        []      xyz:name ?industryName;
                xyz:code ?code .
}
        BIND(IRI(CONCAT(STR(ex:),?companyName)) AS ?company) .
        BIND(IRI(CONCAT(STR(ex:),?industryName)) AS ?industry) .
        BIND(xsd:integer(?revenueString) AS ?revenue) .
        FILTER (?industryCode = ?code) .
}
```

Alternative SERVICE statements using additional variables

```
 SERVICE <x-sparql-anything:location=./data/companies.xml> {
        ?b1 xyz:name ?companyName.
        ?b1 xyz:revenue ?revenueString.
        ?b1 xyz:industryCode ?industryCode
 }
 SERVICE <x-sparql-anything:location=./data/industries.xml> {
        ?b2 xyz:name ?industryName.
        ?b2 xyz:code ?code .
 }
```

Figure 16: SPARQL Anything query for use with XML files in Figures 14 and 15

```
@prefix xyz: <http://sparql.xyz/facade-x/data/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xyz: <http://sparql.xyz/facade-x/data/> .
@prefix fx:  <http://sparql.xyz/facade-x/ns/> .

[ a  xyz:companies , fx:root ;
  rdf:_1    [ a  xyz:company ;
              xyz:industryCode "fi" ;
              xyz:name "AceCo" ;
              xyz:revenue "2420"
            ] ;
  rdf:_2    [ a xyz:company ;
              xyz:industryCode "co" ;
              xyz:name "BuildCo" ;
              xyz:revenue "10010"
            ] ;
  rdf:_3    [ a xyz:company ;
              xyz:industryCode "te" ;
              xyz:name "CableCo" ;
              xyz:revenue "990"
            ]
] .
```

first node in first graph pattern of Figure 16 binds to this node …

… and to this node …

… and to this node.

Figure 17: intermediate ontology created from the XML data of Figure 14

END OF TUTORIAL