Semantic SPARQL Templates

Christina Davril

March 2023

Contents

1 Selection of Set of Items

1.1 all items of type X

```
SELECT DISTINCT ?item ... WHERE {
   ?item <is> wd:Q<num> .
   ...
}
```

<is> is a placeholder for property paths.
In most cases it can be p:P31/ps:P31/(p:P279/ps:P279)* which retrieves instances of (a subclass of (a subclass of ...)) the class characterized by the Q-item with any integer inserted for <num>.
The special case of accessing named entities directly is not covered here.

1.2 ... or Y (or ...)

```
SELECT DISTINCT ?item ... WHERE {
    ...
    { ?item <is> wd:Q<num_1> . }
    UNION
    { ?item <is> wd:Q<num_2> . }
}
```

2 Conditional Selection / Filtering

2.1 ...fulfilling property A (and B (and ...))

2.1.1 ... using property statements

```
1  SELECT DISTINCT ... WHERE {
2      ...
3      ?item p:P<num>/ps:P<num> <prop> .
4  }
```

<prop> can be a URI or a literal.
Note that instead of ps, one can use pq if a statement qualifier must be accessed.
This applies to all templates!

2.1.2 ... using FILTER

```
1  SELECT DISTINCT ... WHERE {
2      ...
3      FILTER(<bool_func>(?attr, ...))
4  }
```

<bool_func> is a boolean function taking the attributes by whose values should be filtered as argument. These attributes can for example be retrieved using property statements, and there can be any number of them.

2.1.3 ... using GROUP BY and HAVING

```
1  SELECT DISTINCT ... WHERE {
2    ...
3  }
4  GROUP BY ?attr, ...
5  HAVING (<bool_func>(?agg_attr, ...))
```

The attribute to group by can be the URI (ID) of the items. It is also possible to group by multiple attributes.
The boolean function used here must contain an aggregated attribute (?agg_attr), as HAVING is used to filter at the group level.
The aggregation of the attribute can happen in the SELECT statement.
?attr and ?agg_attr can be different variables.

2.1.4 ... using ORDER BY and LIMIT and/or OFFSET

```
SELECT DISTINCT ... WHERE {
  ...
}
ORDER BY ASC(?attr, ...)
LIMIT X
OFFSET Y
```

Instead of ASC one can use DESC to reverse the order of sorting. If neither ASC() nor DESC() is specified, the tuples are sorted in ascending order.
LIMIT X filters out everything after the X^{th} entry according to the ordering.
OFFSET Y can be used to filter out the first Y entries in the ordered results list.
This building block can be used in combination with grouping (ordering by an aggregated attribute) or without. It is often used with LIMIT 1 and without any OFFSET to output the top entry according to some attribute ordering.

2.1.5 ... using subqueries (6 variants)

In the following, six variants of templates containing subqueries that were found in the data are presented.

Variant 1

```
1   SELECT DISTINCT ?item ?attr_2 WHERE {
2       ?item <is> wd:Q<num_1> .
3       ?item p:P<num_2>/ps:P<num_2> ?attr_2 .
4       ?item p:P<num_2>/pq:P<num_3> ?agg_attr_1 .
5       {
6           SELECT ?item (<AGG>(?attr_1) AS ?agg_attr_1) WHERE {
7               ?item <is> wd:Q<num_1> .
8               ?item p:P<num_2>/pq:P<num_3> ?attr_1 .
9           }
10          GROUP BY ?item
11      }
12  }
```

Note that DISTINCT is not used in the subquery in case duplicates or their counts are required. Whether DISTINCT is needed in the subquery or not, depends on the concrete example.
The above template shows the most typical use case of a query with subqueries:
The items considered here have a statement group (possibly containing multiple statements) for the property P<num_2>.
Instead of outputting all of these statements, only **statements whose qualifier value is equal to an aggregated value across all values of the statement qualifier** of the item are output (lines 4 and 5).
In most cases, the aggregation function is MAX. The inner query then retrieves the maximum qualifier value for each item (taken across all of its statements).
The outer query then uses this value (?agg_attr_1) to output the statement values (ps) with this qualifier value (pq) – filtering the statements for each of the individual items.

Variant 2

```
SELECT DISTINCT (<AGG>(?agg_attr_1) AS ?agg_attr_2) WHERE {
    {
        SELECT (<AGG>(?attr_1) AS ?agg_attr_1) WHERE {
            ?item <is> wd:Q<num_1> .
            ?item p:P<num_2>/ps:P<num_3> ?attr_1 .
        }
        GROUP BY ?item
    }
}
```

This template uses a subquery because it performs an **aggregation of an attribute at group level** (e.g., grouping by the items) using the inner query, and then a second **aggregation of the (aggregated) attribute** – without any further grouping – in the outer query.
For example, the subquery might retrieve `COUNT`s, while the outer query might calculate and return the `AVG` (average) of all these `COUNT`s.
Since a single aggregated value is output, `DISTINCT` is not used here.
The same is true for the variants to follow. Since we get only one result tuple, `DISTINCT` could be left out.

Variant 3

```
SELECT DISTINCT ?item ?agg_attr_1 WHERE {
    {
        SELECT ?item (<AGG>(?attr) AS ?agg_attr_1) WHERE {
            ?item <is> wd:Q<num_1> .
            ?item p:P<num_2>/ps:P<num_3> ?attr .
        }
        GROUP BY ?item
    }
    BIND(<AGG>(?agg_attr_1) AS ?agg_attr_2)
    FILTER(?agg_attr_1 = ?agg_attr_2)
}
```

This variant is similar to Variant 1, as the subquery template is exactly the same. However, here, the attribute aggregated in the subquery (`?agg_attr_1`) is aggregated a second time (`?agg_attr_1`) – this time over all tuples – and the tuples are filtered by keeping only the ones where the first aggregated value matches the second one.
For example, the inner query could return items with a `COUNT` for some attribute, and

the outer query could then extract the MAXimum value for these counts. The matching using FILTER would then allow to only output items with a maximum count for the attribute. Since BIND defines a new variable, an equality FILTER needs to be used for this.

Variant 4

```
SELECT DISTINCT ?bound_if_attr WHERE {
    {
        SELECT (<AGG>(?attr_1) AS ?agg_attr_1) WHERE {
            ?item <is> wd:Q<num_1> .
            ?item p:P<num_2>/ps:P<num_3> ?attr_1 .
        }
    }
    {
        SELECT (<AGG>(?attr_2) AS ?agg_attr_2) WHERE {
            ?item <is> wd:Q<num_4> .
            ?item p:P<num_2>/ps:P<num_3> ?attr_2 .
        }
    }
    BIND(IF(<bool_func>(?agg_attr_1, ?agg_attr_2),
        ?agg_attr_1, ?agg_attr_2) AS ?bound_if_attr)
}
```

This variant contains **two or more subqueries**, each outputting an aggregation over some attribute. Importantly, the subqueries can be about different sets of items. Technically, the attributes output by the subqueries can also be different ones.
The outer query then uses the combined construct BIND(IF(...)...) to apply a boolean function to the attribute values to be compared. Depending on the result, either the first or second variable is bound to the result variable ?bound_if_attr. Since we get only one result tuple, DISTINCT could be left out.

Variant 5

```
SELECT ?item (<AGG>(?attr_2) AS ?agg_attr_2)
(SAMPLE(?agg_attr_1) AS ?ref) WHERE {
    ?item <is> wd:Q<num_3> .
    ?item p:P<num_4>/ps:P<num_4> ?attr_2 .
    {
        SELECT (<AGG>(?attr_1) AS ?agg_attr_1) WHERE {
            wd:Q<num_1> p:P<num_2>/ps:P<num_2> ?attr_1 .
        }
    }
}
GROUP BY ?item
HAVING (<bool_func>(?agg_attr_2, ?ref))
```

In this variant, the subquery accesses a named entity directly, and retrieves an aggregated value for an attribute of the Q-item.
This value (`?agg_attr_1`) will then serve as a reference value for filtering the values retrieved in the outer query:
The values in the outer query are aggregated values over groups of attributes (e.g., grouped by `?item`). As such, they are filtered using `HAVING`.
In order to use the reference value from the inner query for a comparison at group level, it also needs to be aggregated. This can be done with `SAMPLE`.
While the data did not contain an example for this, one can also image that the inner query could retrieve an aggregated value over any set of tuples – not necessarily a set of property statements for a named entity.
Similarly, one can imagine that the outer query can also contain a non-aggregated attribute that is compared to the reference value using `FILTER`.

Variant 6

```
SELECT DISTINCT (<AGG>(?attr_2) AS ?agg_attr) WHERE {
    {
        SELECT ?item ?attr_1 ?attr_2 WHERE {
            ?item <is> wd:Q<num_1> .
            ?item p:P<num_2>/ps:P<num_2> ?attr_1 .
            ?item p:P<num_3>/ps:P<num_3> ?attr_2 .
        }
        ORDER BY ASC(?attr_1)
        LIMIT X
        OFFSET Y
    }
}
```

In this template, the inner query is used to **filter the values by an attribute using** `ORDER BY` **and** `LIMIT` **and/or** `OFFSET`. The sorting could also be in descending order or use multiple attributes.
The outer query then performs an `aggregation` of an attribute (the same attribute or another) for these filtered tuples.
Note that a nested query is needed because the tuples are first filtered, using this particular method of filtering, and then aggregated.
In comparison, using `FILTER` or simple triples for filtering would not require the use of a nested query.
Since we get only one result tuple, `DISTINCT` could be left out.

2.2 ... fulfilling property A or B (or ...))

This is a variant of 2.1. For simplicity reasons, it is only shown here how this semantics is achieved using property statements.

```
SELECT DISTINCT ... WHERE {
    ...
    { ?item p:P<num_1>/ps:P<num_1> <prop_A> }
    UNION
    { ?item p:P<num_2>/ps:P<num_2> <prop_B> }
}
```

If a `FILTER` or `GROUP BY` and `HAVING` is involved, the boolean function can contain `||`-connected conditions for additional properties.
The filtering using `ORDER BY` and `LIMIT` and/or `OFFSET` can only be applied to multiple attributes if subqueries are used. This is, however, an edge case not treated here.

2.3 ... fulfilling either property A or B (or ...))

This is a variant of 2.1. For simplicity reasons, it is only shown here how this semantics is achieved using property statements.

```
SELECT DISTINCT ... WHERE {
    ...
    { ?item p:P<num_1>/ps:P<num_1> <prop_A> }
    UNION
    { ?item p:P<num_2>/ps:P<num_2> <prop_B> }
    MINUS
    {
        ?item p:P<num_1>/ps:P<num_1> <prop_A> .
        ?item p:P<num_2>/ps:P<num_2> <prop_B> .
    }
}
```

As in the previous subsection, one can extend `FILTER` and `HAVING` functions to the same effect.

2.4 ... not fulfilling property A (and B (and ...))

This is a variant of 2.1. For simplicity reasons, it is only shown here how this semantics is achieved using property statements.
'not fulfilling property A and B' is understood as neither fulfilling property A nor B. We, thus, use a `MINUS` section for each such property.

```
1  SELECT DISTINCT ... WHERE {
2      ...
3      MINUS
4      {
5          ?item p:P<num>/ps:P<num> <prop> .
6      }
7  }
```

3 Output Configuration

3.1 ... listed with attribute Y (and Z (and ...))

```
SELECT DISTINCT ... ?list_attr WHERE {
    ...
    OPTIONAL { ?item p:P<num>/ps:P<num> ?list_attr }
}
```

Since the set of items is not filtered by the attribute, but the attribute is merely output, `OPTIONAL` is used in case there are missing values.

3.2 ... ordered by attribute Y (and Z (and ...))

```
SELECT DISTINCT ... ?sort_attr WHERE {
    ...
    ?item p:P<num>/ps:P<num> ?sort_attr .
}
ORDER BY ASC(?sort_attr)
```

Note that `OPTIONAL` is not used for the attribute to sort by. One can also use `DESC` to sort in reverse order, as well as sort by multiple attributes.
Further modifications of the output attribute values, such as grouping by items and using `GROUP_CONCAT` on the listed attributes, are possible.
We can do everything listed in Section 2 of this document, without needing to use the modifications for filtering.