Further with Knowledge Graphs. M. Alam et al. (Eds.)
AKA Verlag and IOS Press, 2021

doi:10.3233/SSW210047

Adding Structure and Removing Duplicates in SPARQL Results with Nested Tables

Sébastien FERRÉ [a,1]
[a] *Univ Rennes, CNRS, IRISA*
Campus de Beaulieu, 35042 Rennes, France
Email: ferre@irisa.fr

AbstractThe results of a SPARQL query are generally presented as a table with one row per result, and one column per projected variable. This is an immediate consequence of the formal definition of SPARQL results as a sequence of mappings from variables to RDF terms. However, because of the flat structure of tables, some of the RDF graph structure is lost. This often leads to duplicates in the contents of the table, and difficulties to read and interpret results. We propose to use nested tables to improve the presentation of SPARQL results. A nested table is a table where cells may contain embedded tables instead of RDF terms, and so recursively. We introduce an automated procedure that lifts flat tables into nested tables, based on an analysis of the query. We have implemented the procedure on top of Sparklis, a guided query builder in natural language, in order to further improve the readability of its UI. It can as well be implemented on any SPARQL querying interface as it only depends on the query and its flat results. We illustrate our proposal in the domain of pharmacovigilance, and evaluate it on complex queries over Wikidata.

Keywords. Semantic Web, RDF Graph, SPARQL, Presentation of Results, Nested Table, Sparklis

1. Introduction

The SPARQL query language offers a powerful way to extract and compute information from RDF datasets. For the most common form of queries, `SELECT` queries, the results are structured and displayed as a table. Each column corresponds to a projected variable in the `SELECT` clause, and each row corresponds to a query answer, i.e. a mapping from those variables to RDF terms. Such tables have many advantages. They are universally understood. They can be read in two directions, by row or by column. They make a good use of the screen space, compared for instance to graph visualizations, and can therefore display a lot of information at once. They can be dynamically filtered and ordered according to each column. It is no surprise that they are found in all display and visualization frameworks, from HTML to D3js.

Despite all those advantages, tables in general and SPARQL results in particular have drawbacks due to the fact that they are in first normal form (1NF). 1NF is a notion from relational databases [1] that states that table cells only contain atomic values, RDF terms in the case of

[1]This research is supported by ANR project PEGASE (ANR-16-CE23-0011-08).

SPARQL results. It sounds like a reasonable constraint but it has negative consequences on the readability of query results. Imagine that a user wants to retrieve in a touristic dataset all hotels with rating 4 or more in Roma. She also wants to know for each hotel the available services and their cost, as well as reviews on that hotel. Because of the structure of SPARQL results, there will be a row for each combination of a hotel, an available service at that hotel, and a review on that hotel. If on average there are 10 services and 10 reviews per hotel, there will be on average 100 rows per hotel in the table of results, each repeating the same hotel name. Each service and review will also be repeated 10 times for each hotel. All those duplicates hinder the readability of the results. Moreover, the RDF graph structure where services and reviews are related to the hotel but not to each other is lost because of the flat structure of the table.

We propose an automated procedure to post-process SPARQL results into *nested tables* in order to address the previous drawbacks while retaining the advantages of tables. A nested table relaxes the 1NF constraint by allowing each table column to contain either RDF terms or nested tables, whose columns can in turn contain RDF terms or nested tables, and so on. In the above example about hotels, a nested table would have a single row per hotel, the service column would contain two-columns tables listing services with their cost, and the review column would contain one-column tables listing customer reviews.

The automated nesting procedure is decomposed into two algorithms. The first algorithm analyzes a SPARQL query to identify a *nested schema* describing the structure of the nested table to be produced. The second algorithm converts a standard table of results into a nested table, given the nested schema produced by the first algorithm. Together, those algorithms allow for the computation of nested tables with existing query engines, and without any additional requirement from users.

Automated nesting has been implemented in Sparklis[2] [2], a guided query builder that completely hides SPARQL queries behind a natural language interface. This improves further the usability of Sparklis by offering a more natural presentation of results. However, the nesting procedure can also be implemented on any SPARQL querying interface as it only depends on the query and its flat results.

The paper is organized as follows. Section 2 discusses related work. Section 3 defines nested schemas and nested tables in contrast to flat tables, and introduces two illustrative example queries and results. Section 4 presents the main contribution of this paper, the automated procedure from standard queries and flat tables to nested tables. Section 5 describes our implementation, and illustrates it with an application in pharmacovigilance. Section 6 presents a qualitative evaluation of our approach on a range of complex queries over Wikidata. Section 7 concludes and draws some perspectives.

2. Related Work

Nested tables were proposed and formalized in relational databases, where nesting is not only used for query results but also for data tables [3,4]. The motivation was to relax the first normal form (1NF) that states that only atomic values are allowed in table cells. The authors extend relational algebra with two operators: *nest* and *unnest*. To nest a table is to group a subset of columns into one higher-order column, whose values become nested tables. It is a form of aggregation, and it generally produces a table with fewer rows. To unnest a table is to explode a higher-order column into the set of columns it is made of. It is the inverse of the nest operation, and it generally produces a table with more rows. The authors also introduce a Partitioned Normal Form (PNF) that ensures that every nesting has an inverse unnesting by excluding some form of redundancy. Querying nested relational databases involves the application of the usual operations of relational algebra [1] on nested tables, plus nesting and unnesting operations.

[2]Available online at http://www.irisa.fr/LIS/ferre/sparklis/

Linked Data Query Wizard [5] is a tabular interface for the semantic web, where tables have a single row per subject, and possibly several values in cells. However, those tables are limited to the presentation of the direct neighborhood of entities (columns are entity properties, and column values are the objects of those properties) rather than results of arbitrary queries. Table cells can contain sets of values but not multi-column tables like in nested relational databases and what we propose in this paper. A lot of work have proposed various forms of visualization of SPARQL results in order to help their understanding [6,7,8,9]. Those visualizations generally include maps, charts, timelines, etc. They are definetely useful, especially for numerical data. However we think that they are a complement to a tabular view, and that they cannot fully replace it in the general case. A more general approach consists in supporting the transformation of SPARQL results into a broad set of data structures, e.g. using JSON as output [10], where nested tables are one possible output. However, this approach targets web developers, and does not fit end-users that can freely build their own queries, like with Sparklis [2].

Some features of SPARQL allow to structure query results to some degree. `CONSTRUCT` queries produce RDF graphs from query answers, hence offering a lot of flexibility in the produced structure but the concrete layout of the graph is left unspecified. A `GROUP_CONCAT` aggregate enables to pack a set of RDF terms into a single RDF term but this requires to convert all terms to strings, and the internal structure of aggregate values is lost, unlike with nested tables. Some extensions of SPARQL have also been proposed. For instance, a new `CLUSTER BY` clause was proposed to group the rows of the table of results into a hierarchical clustering [11]. In contrast, nested tables can be seen as a form of 2D hierarchical clustering because they involve grouping subsets of columns and subsets of rows at the same time. In a previous work [12], we have proposed to extend SPARQL 1.1 with a new kind of aggregation construct in order to have nested tables as a native SPARQL results. The work presented in this paper brings two new contributions. First, the structure of the nested table is automatically derived from a standard SPARQL query and its results. Second, it allows for the computation of nested results on top of standard SPARQL 1.1 endpoints, hence supporting a larger adoption of nested results, independently of the extension proposed earlier.

3. Motivating Nested Tables

The results of a SPARQL query are traditionally displayed as (flat) tables.

Definition 1 (flat table) *A* flat table *FT is a sequence of* rows*, where each row $FT[i]$ is a partial mapping from* column names *to values. Notation $FT[i].x$ refers to the contents of the cell (a value or nothing) that is at the crossing between the i-th row, and column x.*

In SPARQL results, rows are query answers, columns are SPARQL variables from the `SELECT` clause, and values are RDF terms. The ordering of columns is determined by the `SELECT` clause of the query, and the sequence of rows may be ordered or not depending on the existence of an `ORDER BY` clause in the query.

Table 1 shows an excerpt of the results of the following SPARQL query on DBpedia, which retrieves films directed by Danny Boyle, along with their music composers and actors, and also the birth year of actors[3].

```
SELECT ?f ?mc ?a ?y
WHERE { ?f a dbo:Film ;
           dbo:director dbr:Danny_Boyle ;
           dbo:musicComposer ?mc ;
```

[3]In table headers, variable names have been replaced by more explicit names for readability.

Table 1. Flat table of films by Danny Boyle with music composer, actor, and actor's birth year

film (f)	music composer (mc)	actor (a)	birth year (y)
Slumdog millionaire	A. R. Rahman	Dev Patel	1990
Slumdog millionaire	A. R. Rahman	Freida Pinto	1984
Slumdog millionaire	A. R. Rahman	Anil Kapoor	1959
Sunshine	John Murphy	Cilian Murphy	1976
Sunshine	John Murphy	Chris Evans	1981
Sunshine	John Murphy	Rose Byrne	1979
Sunshine	John Murphy	...	...
Sunshine	Underworld	Cilian Murphy	1976
Sunshine	Underworld	Chris Evans	1981
Sunshine	Underworld	Rose Byrne	1979
Sunshine	Underworld	...	...
...	...	...	...

Table 2. Flat table of countries with their capital, their most populated cities, and for each city, the population and the mathematicians who died in it.

country (c)	capital (cap)	city	population (pop)	mathematician (m)	death date (dd)
United States	Washington	New York	8 550 405	E. L. Post	1954-04-21
United States	Washington	New York	8 550 405	R. Schatten	1977-08-26
United States	Washington	New York	8 550 405	...	...
United States	Washington	Los Angeles	3 792 710	R. Montague	1971-03-07
United States	Washington	Los Angeles	3 792 710	R. E. Bellman	1984-03-19
United States	Washington	Los Angeles	3 792 710	...	...
United States	Washington	Chicago	2 707 120	E. Hellinger	1950-03-28
United States	Washington	Chicago	2 707 120	E. H. Moore	1932-12-30
...	...	...	...	...	...

```
           dbo:starring ?a .
        ?a dbo:birthYear ?y . }
```

It can be observed that the table contains a lot of redundancies. For instance, each actor and his birth year is repeated for each music composer. The number of rows for a film is equal to its number of music composers times its number of actors. When ordering by birth year, one needs to order first by film so as to keep rows grouped by film.

Table 2 shows an excerpt of the results of another SPARQL query that retrieves various information about countries: capital, most populated cities, and mathematicians who died in those cities.

```
SELECT ?c ?cap ?city ?pop ?m ?dd
WHERE { ?c a dbo:Country ;
           dbo:capital ?cap .
        ?city a dbo:City ;
              dbo:country ?c ;
              dbo:populationTotal ?pop .
        ?m a dbo:Scientist ;
           dbo:field dbr:Mathematics ;
           dbo:deathPlace ?city ;
           dbo:deathDate ?dd . }
```

Table 3. Nested table of films by Danny Boyle with music composers, and actors with birth year

film (f)	*music composers*	*actors*	
Slumdog millionaire	music composer (mc)	actor (a)	birth year (y)
	A. R. Rahman	Anil Kapoor	1959
		Freida Pinto	1984
		Dev Patel	1990
		...	...
Sunshine	music composer (mc)	actor (a)	birth year (y)
	John Murphy	Cilian Murphy	1976
	Underworld	Rose Byrne	1979
		Chris Evans	1981
		...	...
...	...	...	

```
ORDER BY ?c DESC(?pop)
```

It can be observed again that the table contains redundancies, here about countries and capitals. More importantly, a lot of structure and dependencies are lost in the flat table. It is not clear, without reading the query, whether the population is about the city or the country, whether the listed mathematicians died in the city or more generally in the country.

The key idea of this paper is to present results in more structured tables, in order to make them more readable by removing redundancies and by exhibiting some of the underlying RDF graph structure.

Definition 2 (nested table and nested schema) *A* nested table *NT is a sequence of rows, where each row $NT[i]$ is a partial mapping from column names to values* or nested tables, *according to a nested schema. A* nested schema *is an ordered forest, i.e. a sequence of ordered trees, such that there is a bijection between the forest nodes and the columns of the nested table. Nested table NT agrees with a nested schema $N = \alpha_1 \dots \alpha_k$ if the following conditions are satisfied for every row i, and every tree α_j:*

1. *if $\alpha_j = x_j$ is a leaf, then $NT[i].x_j$ contains a value (or nothing);*
2. *if $\alpha_j = X_j[N_j]$ is an internal node N_j with children $N_j = \beta_1 \dots \beta_l$, then $NT[i].X_j$ contains a nested table agreeing with nested schema N_j.*

A nested table agreeing with a nested schema follows a regular structure. For a given column, either all rows contain an RDF term or all rows contain a table following the same structure. Note that the definitions of nested tables and nested schemas are recursive, thus allowing arbitrary numbers of nesting levels.

In the following, we represent nested schemas using lowercase names for leaves, uppercase names for internal nodes, and square brackets to delimit the children of an internal node. For example, nested schema `a b C[d E[e]]` has three levels of tables: the outer table has two classical columns *a* and *b*, and a nested column *C*, which contains nested tables with a classical column *d*, and a nested column *E*, which in turn contains nested tables with a single column *e*.

Table 3 is the nested version of Table 1 for the nested schema

$$N_{film} = \texttt{f MC[mc] A[a y]}.$$

The outer table has a single row per film, and two of its columns, "music composers" (`MC`) and "actors" (`A`), contain nested tables. The former column contains one-column nested tables that

Table 4. Nested table of countries with their capital, and their most populated cities. Each city is described by its population, and a list of mathematicians who died there.

country (c)	capital (cap)	*cities*			
		city	population (pop)	*mathematicians*	
United States	Washington	New York	8 550 405	mathematician (m)	death date (dd)
				E. L. Post	1954-04-21
				R. Schatten	1977-08-26
				...	...
		Los Angeles	3 792 710	mathemtician (m)	death date (dd)
				R. Montague	1971-03-07
				R. E. Bellman	1984-03-19
				...	...
		Chicago	2 707 120	mathematician (m)	death date (dd)
				E. H. Moore	1932-12-30
				E. Hellinger	1950-03-28
				...	...
		...	...	...	
...	...	...			

contain the lists of music composers of each film. The latter column contains two-columns nested tables that contain the lists of actors of each film, along with their birth year. It can be observed that the nested table does not contain redundancies anymore, and that the dependencies between columns is made explicit: (a) music composers and actors are dependent on the film, but not on each other; (b) the birth year is dependent on the actor.

Table 4 is the nested version of Table 2 for the nested schema

$$N_{country} = \texttt{c cap CITIES[city pop M[m dd]]}.$$

The outer table has a single row per country, a column "capital" that contains RDF terms, and another column "cities" that contains nested tables, which themselves contain nested tables about the mathematicians who died in the city. The nested table structure clearly shows that countries have one capital but (possibly) several cities, that the population is about the city, and that the listed mathematicians are related to the city rather than to the country.

In the following section, we explain how a nested table can be automatically derived from the flat table, by analyzing the SPARQL query.

4. Automated Nesting of SPARQL Results

We here consider the computation of a nested table from the flat table resulting from the evaluation of a standard SPARQL query. Section 4.1 describes an algorithm *NestedSchema* that extracts a nested schema from the query. Section 4.2 describes an algorithm *NestedTable* that restructures the query results into a nested table agreeing with a given nested schema. By combining the two algorithms, it becomes possible to get nested tables as the result of a standard query evaluated on a standard SPARQL 1.1 engine. Given a query Q, this can be summarized by the definition

$$NestedEval(Q) = NestedTable(Eval(Q), NestedSchema(Q))$$

where *Eval* is standard SPARQL evaluation.

```
SELECT ?f ?mc ?a ?y
WHERE { ?f a dbo:Film ;
           dbo:director dbr:Danny_Boyle ;
           dbo:musicComposer ?mc ;
           dbo:starring ?a .
        ?a dbo:birthYear ?y . }
```

Figure 1. Example query on films.

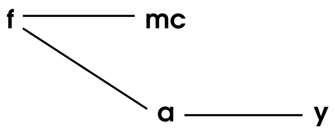


Figure 2. Dependency graph of the query in Figure 1 (ignoring non-connected vertices).

4.1. Nested Schema of a Query

Our procedure to compute the nested schema of a query relies on the dependencies between variables expressed by the SPARQL query, and on an ordering of those variables. We first explain the main principles of the automated procedure on the example about films, whose query we recall in Figure 1 for convenience. The expected nested schema is here N_1 = `f MC[mc] A[a y]`. It has two nesting levels (tables in a table). Nesting N_0 = `f mc a y` represents a flat table with no nesting (one level). Nesting N_2 = `f MC[mc A[a Y[y]]]` represents a deep nesting with 4 levels. The reason why N_1 is better than N_0 and N_2 is that music composers and actors are related to films but not to each other. In other words, the set of music composers and the set of actors depend on the film but given a film, the set of actors does not depend on a particular music composer.

Dependencies between variables can be infered from the query abstract syntax. In short, two variables are dependent on each other if they occur in the same triple pattern, e.g. variables `f` and `mc` in the example. More generally, we formally define the dependency graph of an arbitrary SPARQL query.

Definition 3 (dependency graph) *The* dependency graph *of a SPARQL query Q is the undirected graph $G(Q) = (V,E)$, where the set of vertices V is the set of variables occuring in the query, and where the set of edges $E \subseteq V \times V$ is the set of dependencies $deps(g_0,Q)$, defined inductively on the syntax of the query according to the following equations, starting with the default graph g_0.*

$$deps(g, \texttt{SELECT}\ X\ \texttt{WHERE}\ P) = deps_X(X) \cup deps_P(g,P)$$

$$deps_X(x_1\ \ldots\ x_n) = \emptyset$$
$$deps_X(x_1\ \ldots\ x_n\ (A_1(y_1)\ \texttt{AS}\ z_1)\ \ldots\ (A_k(y_k)\ \texttt{AS}\ z_k)) = \{(x_i,z_j) \mid i \in [1,n], j \in [1,k]\}$$

$$deps_P(g, s\ p\ o) = \{(x,y) \mid x,y \in Vars(\{s,p,o,g\}), x \neq y\}$$
$$deps_P(g, P_1\ .\ P_2) = deps_P(g,P_1) \cup deps_P(g,P_2)$$
$$deps_P(g, P_1\ \texttt{UNION}\ P_2) = deps_P(g,P_1) \cup deps_P(g,P_2)$$
$$deps_P(g, P_1\ \texttt{OPTIONAL}\ P_2) = deps_P(g,P_1) \cup deps_P(g,P_2)$$
$$deps_P(g, P_1\ \texttt{MINUS}\ P_2) = deps_P(g,P_1)$$
$$deps_P(g, P\ \texttt{FILTER}\ E) = deps_P(g,P)$$
$$deps_P(g, \texttt{BIND (}E\ \texttt{AS}\ x\texttt{)}) = \{(y,x) \mid y \text{ a variable occuring in } E\}$$
$$deps_P(g, \texttt{GRAPH}\ g'\ P) = deps_P(g',P)$$
$$deps_P(g, \texttt{SERVICE}\ s\ P) = deps_P(s.g_0,P)$$
$$deps_P(g, \texttt{VALUES}\ (x_1\ \ldots\ x_n)\ data) = \{(x_i,x_j) \mid 1 \leq i < j \leq n\}$$
$$deps_P(g, \{\ Q\ \}) = deps(g,Q)$$

The inductive definition goes through queries ($deps(g,Q)$), projections ($deps_X(X)$), and graph patterns ($deps_P(g,P)$), given a current graph g: g_o is the default graph, and $s.g_0$ is the default graph of service s. Letter A denotes an aggregate operator (e.g. `COUNT`). Expressions in the `SELECT` and `GROUP BY` clauses are assumed to be rewritten as `BIND` clauses to simplify the definition. Solution modifiers (e.g., `ORDER BY, LIMIT`) do not produce dependencies.

Algorithm 1 $NestedSchema(G,X)$

```
Require: $G = (V,E)$ a connected dependency graph
Require: $X$ a sequence ordering variables in $V$
Ensure: $N$ a nested schema whose set of leaf columns is $V$
 1: if $|X| = 1$ then
 2:     $N \leftarrow X$
 3: else
 4:     $x, Y \leftarrow head(X), tail(X)$
 5:     $G_x \leftarrow (V \setminus \{x\}, E \setminus (\{x\} \times V) \setminus (V \times \{x\}))$
 6:     $G_1, \ldots, G_k \leftarrow ConnectedComponents(G_x)$
 7:     $Y_1, \ldots, Y_k \leftarrow$ partition of $Y$ according to $G_1, \ldots, G_k$
 8:     $N_1, \ldots, N_k \leftarrow NestedSchema(Y_1, G_1), \ldots, NestedSchema(Y_k, G_k)$
 9:     $N \leftarrow x\ X_1[N_1]\ \ldots\ X_k[N_k]$ {w}here each $X_i$ is a fresh nested column name
10: end if
```

Figure 2 shows the dependency graph of the example query. Dependencies come mostly from the graph pattern in the `WHERE` clause, and also from aggregates. In triple patterns, each variable depends on other variables, including the current graph g if different from the default graph (quads). Then the dependencies from the different sub-patterns are collected, except for those at the right of `MINUS` as they do not produce any binding. Expressions in `FILTER` do not produce any dependency but in `BIND`, they produce a dependency from every used variable to the bound variable as the value of the latter depends on the value of each of the former. Similarly, every aggregate variable in the `SELECT` clause depends on each of the non-aggregate variables, i.e. on the grouping variables. In the `GRAPH` and `SERVICE` constructs, we produce the dependencies of the embeded graph pattern, only changing the current graph. In the `VALUES` clause, we have no information about the dependencies between the bound variables, and we have to consider that they are all mutually dependent. Indeed, the consequence of ignoring a dependency (a loss of information) is greater than considering a superfluous dependency (redundancy in results). However, such clauses are rare in practice. Finally, the dependencies of sub-queries are defined like for whole queries.

Algorithm 1 describes the recursive process of computing a nested schema given the connected dependency graph of query (see discussion below for non-connected dependency graphs), and given an ordering over the dependency graph vertices, i.e. the query variables. By default, we take the ordering of variables from their order of introduction in the graph pattern of the query. The `SELECT` clause can be used to alter this ordering. If a variable does not occur in the `SELECT` clasuse, it is pruned from the nested schema as a post-processing, which amounts to remove a leaf node from the forest sructure of the nested schema.

The base case of the algorithm is when there is a single variable. Otherwise, the first variable is used as a key (grouping), and other variables are organized into one or several nested tables. To determine the partitioning of the other variables into different nested schemas, we rely on the dependency graph. First, we remove the first variable x and its adjacent edges from the dependency graph. This may split the dependency graph into several connected components, which determine the partitioning of the remaining variables Y. The procedure is called recursively on each connected component G_i and its associated subsequence of variables Y_i.

In the unusual case where the dependency graph of a query is not connected, i.e. it has several connected components, we run the algorithm on each connected component, and get a set of nested schemas $N_1, \ldots, N_k$. From there, we generate the nested schema $N = X_1[N_1]\ \ldots\ X_k[N_k]$, which has no key variable, and hence specifies a main table with a single row. This implies that the query results are organized as a sequence of independent tables. Indeed, evaluating a disconnected query is equivalent to evaluating several queries, and hence to get several tables of results.

Applying Algorithm 1 on the example about films returns nesting schema

Algorithm 2 *NestedTable*(FT,N)

```
Require: $FT$ a flat table with columns $X$
Require: $N$ a nested schema over a subset of $X$
Ensure: $NT$ a nested version of $FT$
 1: if $N = x$ {n}o nested table then
 2:     $NT \leftarrow$ projection of $FT$ on column $x$, and duplicate removal
 3: else
 4:     $N = x\, X_1[N_1]\ \dots\ X_k[N_k]$
 5:     $\{FT_1,\dots,FT_n\} \leftarrow$ grouping the rows of $FT$ by column $x$ ($n$ groups)
 6:     $NT \leftarrow$ a table with $1+k$ columns $(x, X_1,\dots,X_k)$, and $n$ rows
 7:     for $i = 1$ to $n$ do
 8:         $NT[i].x \leftarrow FT_i[1].x$ {r}etrieving key value in first row of $FT_i$
 9:         for $j = 1$ to $k$ {f}or each nested column do
10:             $NT[i].X_j \leftarrow NestedTable(FT_i, N_j)$ {f}ill cells with nested tables
11:         end for
12:     end for
13:     for $j = 1$ to $k$ {s}implification for empirical functional dependencies do
14:         if $NT[i].X_j$ has a single row for all $i \in [1,n]$ then
15:             replace the nested column $X_j[N_j]$ by the sequence of columns $N_j$
16:         end if
17:     end for
18: end if
```

N_3 = `f MC[mc] A[a Y[y]]`. Note that N_3 is close to the expected nesting schema N_1 but not equivalent. The independency between music composers and actors has been identified but nested tables are introduced for actor's birth year inside the nested table of actors. The reason why this additional nested table is not relevant is that there is a functional dependency from actors to birth years as people are only born once. Unfortunately, the query does not provide this information. If there is access to the ontology or the data, it may be possible to get this information. In the following section, we show how the nested schema can be simplified during the nesting of flat tables by discovering such functional dependencies empirically. To give another example, the nested schema computed by our algorithm on the query about countries is `c CAP[cap] CITIES[city POP[pop] M[m DD[dd]]]`, which is close to the expected nesting `c cap CITIES[city pop M[m dd]]` except for the additional nesting of functional properties (country's capital, city's population, and mathematician's death date).

The variable ordering has an important impact on the computed nested schema. Indeed, there is not a single good choice of nested schema given a dependency graph, and the variable ordering can be used to control which nested schema is produced. From the dependency graph in Figure 2, we obtain the different following nested schemas according to different variable orderings:

- `a y f mc` $\Longrightarrow$ `a Y[y] F[f MC[mc]]`: view by author,
- `mc f a y` $\Longrightarrow$ `mc F[f A[a Y[y]]]`: view by music composer,
- `f a mc y` $\Longrightarrow$ `f A[a Y[y]] mc`: another view by film, actors first.

4.2. Nesting a Flat Table into a Nested Table

The previous section explains how to automatically extract a nested schema N from a query Q. This provides all the necessary information to automatically lift the flat table $FT = Eval(Q)$ into a nested table agreeing with N.

Algorithm 2 computes such a nested table NT given the flat table FT and the nested schema N computed from the query (see Section 4.1). The base case (Lines 1-2) is when the nesting schema has no nested table, which implies it has the form $N = x$ for some variable x. The returned nested table is then column x of the flat table, in which duplicates are removed (like in a `SELECT DISTINCT ?x`). In the general case (Lines 3-18), the nesting has the form $N = x\ X_1[N_1]\ \ldots\ X_k[N_k]$. Such a nesting involves a grouping of the rows of FT according to x (Line 5), like a clause `GROUP BY ?x`, and the computations of the k nested tables on each group (Lines 7-12). Each nested column $X_j[N_j]$ involves a recursive call on each group of rows FT_i with nesting N_j. A drawback of the nested schemas computed by Algorithm 1 is that it does not take into account functional dependencies. In the example about films, the computed nesting is `f MC[mc] A[a Y[y]]`, which implies that in the nested tables about actors, the second column about actor's birth year is filled with one-column one-row tables. Lines 13-17 replaces those one-row nested tables by their single row contents, but only in the nested columns where all nested tables have a single row. This condition identifies *empirical functional dependencies*, i.e. functional dependencies that exist in the flat table but that may not be true in general. In the example, this results in the effective nested schema `f MC[mc] A[a y]`.

5. Implementation and Application

The automated procedure to lift a flat table into a nested table has been implemented in the Sparklis[4][5] querying tool [2]. Sparklis is a SPARQL query builder that hides SPARQL behind a natural language interface, and guides the query construction so that the user does not need to know the schema of the RDF dataset. It works as follows. After each user interaction, the abstract representation of the query is updated, from which a new SPARQL query is derived. Then, the SPARQL query is sent to the SPARQL endpoint, which returns results as a set of mappings from variables to RDF terms. Originally, Sparklis displays those results as a flat table, only improving the display of RDF terms with labels and media contents. The new nesting procedure is applied to the generated SPARQL query and returned results, and feeds the final display process. Currently, Sparklis does not generate names for the nested columns, and their headers are left empty. In the future, those names could be generated in a way similar to classical columns whose names are derived from the labels of the classes and properties applying to the corresponding variable.

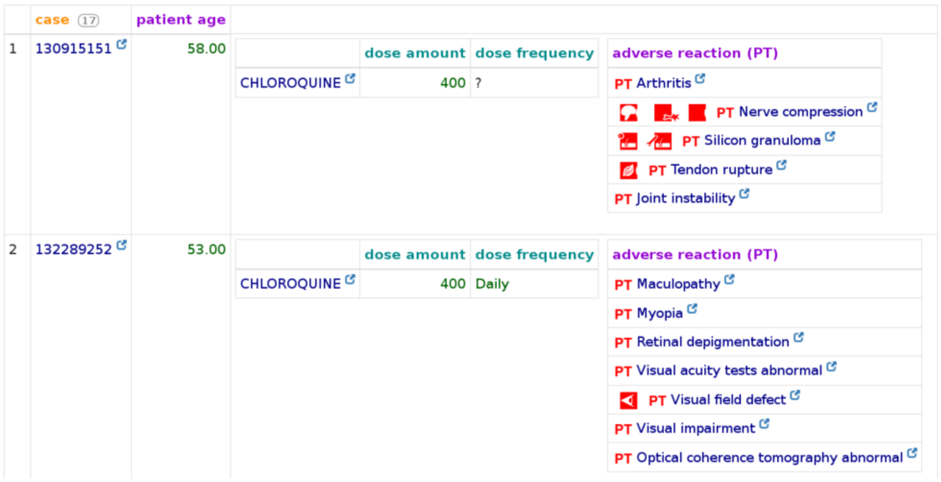
The display of results as nested tables was initially motivated by a use case in pharmacovigilance[6], where domain experts typically look for patient cases depending on the drugs they took, and the adverse reactions they suffered. In this use case, each patient case may have a number of different drugs and adverse reactions. In a flat table, the description of a patient case is therefore scattered over several rows, and a lot of duplicates occur when there are several drugs and several adverse reactions. In a previous user study [13], it was observed that this made it difficult for pharmacovigilance experts to read and interpret the results.

Figure 3 shows a screenshot of the first two rows of the nested table that describes patient cases who suffered adverse reactions to chloroquine. It can be observed that there is a single row per patient, and that this row is well organized. For each patient, the table shows its case id, its age, and two nested tables. The first nested table describes the drug event, giving the dose amount, and optionally the dose frequency. The second nested table gives the list of adverse reactions of the patient. The icons on the left of adverse reactions are taken from an iconic language for the graphical representation of medical concepts [14]. The anonymised patient data are a three-months sample from the FAERS database [15].

[4]Online application at http://www.irisa.fr/LIS/ferre/sparklis/

[5]Open source code at https://github.com/sebferre/sparklis

[6]This is part of ANR research project PEGASE. The dataset is not made available because of property rights on medical ontologies like SNOMED-CT and MedDRA.



	case (17)	patient age		dose amount	dose frequency	adverse reaction (PT)
1	130915151	58.00	CHLOROQUINE	400	?	PT Arthritis
						PT Nerve compression
						PT Silicon granuloma
						PT Tendon rupture
						PT Joint instability
2	132289252	53.00	CHLOROQUINE	400	Daily	PT Maculopathy
						PT Myopia
						PT Retinal depigmentation
						PT Visual acuity tests abnormal
						PT Visual field defect
						PT Visual impairment
						PT Optical coherence tomography abnormal

Figure 3. Screenshot of the nested table showing two patient cases (out of 17) with adverse reactions to chloroquine (in excerpt of FAERS data).

6. Evaluation

We evaluate our automated nesting procedure on a range of complex Wikidata queries, picked and adapted from an online list of Wikidata example queries[7]. We only pick queries with at least three different entities in the `SELECT` clause as otherwise, there is only one possible nested schema: $x\,Y(y)$, possibly unnested into $x\ y$ in case of functional dependency from x to y. Note that we do not count projected variables like `?xxxLabel` as they only serve to capture the label of `?xxx`. We also modify some queries by removing aggregates like `SAMPLE` and `COUNT`. Indeed, such aggregates replace sets of RDF terms by a single RDF term, and hence often remove the need for nested tables. For example, instead of retrieving one sample actor for each film, we retrieve a list of actors for each film.

For each of the nine queries below, we give:

(a) a title,
(b) a short URL[8] that opens the query in Sparklis and enables to visualize the results,
(c) an informal description of the information retrieved by the query,
(d) the obtained nested schema, in which we omit the nested column names, and replace the square brackets by round brackets in the case of unnesting due to a functional dependency in data,
(e) comments on the nested schema, and possibly alternative nested schemas based on a different column ordering.

The queries have been constructed interactively in Sparklis. Their verbalization in English is not always fully satisfying because it relies on the assumption that class and property labels are nouns, which is often but not always verified in Wikidata. However, we believe that those verbalizations are understandable in most cases. The ordering of variables used in the nesting procedure is defined by the order in which they appear (implicitly) in the verbalization.

[7]https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples
[8]Real URLs and query previews are also available at http://www.irisa.fr/LIS/ferre/sparklis/examples.html, section "Nested results on Wikidata."

1. **Academy awards** (`http://rebrand.ly/xyw7nwf`): this query looks for people who received an Academy award for some work. It also retrieves work directors, if any, and the edition and time of the award

   N_1 = `person [work [director] [award (awardEdition (time))]]`

   Each person may have several awarded works. Each work may have several directors on one hand, and several awards (for the same person) on the other hand. According to results, each award belongs to a single Academy awards edition, which occurs on a single time. If the award column is put before the work column, one gets the less interesting nested schema `person [award (work [director] (awardEdition (time)))]`, where the grouping of awards by work is lost and a same work may be repeated across different awards.
2. **Ranking of film directors** (`http://rebrand.ly/qaelfdg`): this query retrieves film directors, counts their number of directed films, and number of cast roles, and rank them by decreasing sum of the two numbers.

   N_2 = `director (filmNb (roleNb (total)))`

   This query illustrates the fact that aggregates and expressions create functional dependencies, in which case there is not practical need for nesting.
3. **The Simpsons episodes** (`http://rebrand.ly/r2b4931`): this query retrieves all seasons and episodes of The Simpsons television series.

   N_3 = `seasonOrdinal (season [episodeOrdinal (episode)])`

   Each season ordinal number (1, 2...) determines a single season. Each season has a number of episodes, identified by their ordinal number in the season. By ordering by season ordinal and episode ordinal, we get a well-structured presentation of all episodes by season.
4. **Law & Order episodes** (`http://rebrand.ly/ib18whw`): this query retrieves all seasons and episodes of the Law & Order series, including publication dates for each episode.

   N_4 = `seasonOrd (season [episodeOrd (episode [pubDate]))`

   We get a similar nested schema compared to the previous query, except we get an additional nesting level for the episode publication dates. Indeed, many episodes have several publication dates.
5. **Dr Who performers** (`http://rebrand.ly/dpxfonc`): this query retrieves all regenerations of Dr Who with their ordinal and performers.

   N_5 = `doctor (ordinal) [performer]`

   The nested schema tells us that each regeneration of Dr Who has a single ordinal but several performers for the First Doctor. The ordinal enables to sort the results in chronological order.
6. **Movies about World War II** (`http://rebrand.ly/hv3hw1p`): this query retrieves narrative locations and countries of origins of films about WWII.

   N_6 = `film [narrativeLoc] [country]`

   The nested schema tells us narrative locations and countries of origin are independant, and that in general, there are several values of each for a given film. An alternative nested schema, `narrativeLoc [film [country]]`, provides WWII films grouped by narrative location.
7. **Longest rivers** (`http://rebrand.ly/xklfe8n`): this query retrieves rivers in decreasing length order, along with pictures of them.

$N_7 =$ `river [length] [picture]`

Surprisingly, it appears that some rivers have several lengths in their description. This typically happens when there are several sources of information. A nice consequence of having nested tables in Sparklis is that, when a river has several pictures, they are displayed as a kind of gallery in a table cell, rather than repeating the whole river row for each picture.

8. **Stations of Paris metro line 1** (http://rebrand.ly/iolf4h3): this query retrieves stations of Paris metro line 1, and for each station, it retrieves the connecting lines if any on one hand, and adjacent stations with directions on the other hand.

   $N_8 =$ `station [line] [adjacent [adjacentLine [adjacentDir]]]`

   The four-levels nested schema tells us that a station my have several connecting lines, what is not surprising. What is more surprising is that some adjacent stations can be reached through several lines, and sometimes for a given line, with several line directions. This happens because some metro lines have tree-shaped ends.

9. **Governments of countries** (http://rebrand.ly/nlp8nex): this query retrieves, for each country, the *current* forms of government (e.g., republic, parliamentary monarchy), and the heads of government since 2000 with their gender and start date. The current forms of government are selected by excluding those having an end date. We also retrieve the start date when available.

   $N_9 =$ `country [form (formStart)] [head (gender) [headStart]]`

   The nested schema shows that forms of government and heads of government are independent, as expected. Heads have a single gender but may have several start dates, corresponding to different mandates.

The above examples show a diverse range of nested schemas, which exhibit structures coming from the RDF graph and left implicit in flat tables. Those structures are often as one would expect but sometimes they reveal unexpected patterns or irregularities in the data: e.g., the fact that metro lines may have several directions through a same adjacent station, or the fact that some rivers have several competing lengths. The above examples also demonstrate the importance of empirical functional dependencies to simplify the structure of nested tables, in particular to reduce the number of nesting levels.

7. Conclusion

We have proposed an automated procedure to re-structure the flat results of a SPARQL query into a nested table. Nested tables improve the readability of results by avoiding redundancies in their contents, and by exhibiting the dependencies and independencies between their columns. The procedure is decomposed into two algorithms. Algorithm *NestedSchema* defines a nested schema given dependencies between variables and an ordering over those variables, both computed from the query. Algorithm *NestedTable* uses that nested schema to restructure the flat table of results into a nested table. One perspective is to generate nested schemas without giving a variable ordering, e.g. by finding a variable ordering that minimizes redundancies and maximizes readability.

References

[1] Codd EF. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM. 1970;13(6):377-87.

[2] Ferré S. Sparklis: An Expressive Query Builder for SPARQL Endpoints with Guidance in Natural Language. Semantic Web: Interoperability, Usability, Applicability. 2017;8(3):405-18. Available from: http://www.irisa.fr/LIS/ferre/sparklis/.

[3] Scholl M, Abiteboul S, Bançilhon F, Bidoit N, Gamerman S, Plateau D, et al. VERSO: A database machine based on nested relations. In: Work. Theory and Applications of Nested Relations and Complex Objects. Springer; 1987. p. 27-49.

[4] Roth MA, Korth HF, Silberschatz A. Extended algebra and calculus for nested relational databases. ACM Transactions on Database Systems (TODS). 1988;13(4):389-417.

[5] Hoefler P, Granitzer M, Sabol V, Lindstaedt S. Linked Data Query Wizard: A Tabular Interface for the Semantic Web. In: The Semantic Web: ESWC 2013 Satellite Events. Springer; 2013. p. 173-7.

[6] Bikakis N, Sellis T. Exploration and visualization in the web of big linked data: A survey of the state of the art. In: Int. Work. Linked Web Data Management (LWDM); 2016. .

[7] Atemezing GA, Troncy R. Towards a Linked-Data based Visualization Wizard. In: Hartig O, Hogan A, Sequeda JF, editors. Int. Work. Consuming Linked Data (COLD), co-located Int. Semantic Web Conf. (ISWC). vol. 1264 of CEUR Workshop Proceedings. CEUR-WS.org; 2014. Available from: http://ceur-ws.org/Vol-1264/cold2014_AtemezingT.pdf.

[8] Skjæveland MG. Sgvizler: A javascript wrapper for easy visualization of sparql result sets. In: Extended Semantic Web Conference. Springer; 2012. p. 361-5.

[9] Martin M, Abicht K, Stadler C, Ngonga Ngomo AC, Soru T, Auer S. Cubeviz: Exploration and visualization of statistical linked data. In: Int. Conf. World Wide Web. ACM; 2015. p. 219-22.

[10] Lisena P, Meroño-Peñuela A, Kuhn T, Troncy R. Easy web API development with SPARQL transformer. In: Int. Semantic Web Conf. Springer; 2019. p. 454-70.

[11] Ławrynowicz A. Query results clustering by extending SPARQL with CLUSTER BY. In: OTM Confederated Int. Conf. on the Move to Meaningful Internet Systems. Springer; 2009. p. 826-35.

[12] Ferré S. A Proposal for Nested Results in SPARQL. In: Taylor K, Gonçalves R, Lecue F, Yan J, editors. ISWC 2020 Posters, Demos, and Industry Tracks. vol. 2721 of CEUR Workshop Proceedings; 2020. p. 114-9. Available from: http://ceur-ws.org/Vol-2721/paper527.pdf.

[13] Marcilly R, Douze L, Ferré S, Audeh B, Bobed C, Lillo-Le-Louët A, et al. How to interact with medical terminologies? Formative usability evaluations comparing three approaches for supporting the use of MedDRA by pharmacovigilance specialists. BMC Medical Informatics and Decision Making. 2020;20(261).

[14] Lamy JB, Duclos C, Bar-Hen A, Ouvrard P, Venot A. An iconic language for the graphical representation of medical concepts. BMC medical informatics and decision making. 2008;8(1):16.

[15] FDA Adverse Event Reporting System (FAERS);. Available from: https://open.fda.gov/data/faers/.