# Analytical Queries on Vanilla RDF Graphs with a Guided Query Builder Approach

Sébastien Ferré

# Analytical Queries on Vanilla RDF Graphs with a Guided Query Builder Approach

Sébastien Ferré*

Univ Rennes, CNRS, IRISA
Campus de Beaulieu, 35042 Rennes cedex, France
Email: ferre@irisa.fr

**Abstract.** As more and more data are available as RDF graphs, the availability of tools for data analytics beyond semantic search becomes a key issue of the Semantic Web. Previous work require the modelling of data cubes on top of RDF graphs. We propose an approach that directly answers analytical queries on unmodified (vanilla) RDF graphs by exploiting the computation features of SPARQL 1.1. We rely on the `N<A>F` design pattern to design a query builder that completely hides SPARQL behind a verbalization in natural language; and that gives intermediate results and suggestions at each step. Our evaluations show that our approach covers a large range of use cases, scales well on large datasets, and is easier to use than writing SPARQL queries.

## 1 Introduction

Data analytics is concerned with groups of facts whereas search is concerned with individual facts. Consider for instance the difference between *Which films were directed by Tim Burton?* (search) and *How many films were produced each year in each country?* (data analytics). Data analytics has been well studied in relational databases with data warehousing and OLAP [2], but is still in its infancy in the Semantic Web [3,9,15]. Most of the existing work adapts the OLAP approach to RDF graphs. Typically, data administrators first derive data cubes from RDF graphs by specifying what are the observations, the dimensions, and the measures [4]. End-users can then use traditional OLAP-based user interfaces for cube transformations and visualizations. More recently, other kinds of user interfaces have been proposed to ease analytical querying: natural language interfaces [16,9,1]; guided query construction, for instance based on pre-defined query templates [11]; and high-level query languages that can be translated to SPARQL [13]. The main drawback of using data cubes is that end-users have no direct access to the original RDF graphs, and can only explore the cubes that have been defined by some data administrator. This drawback is mitigated in [3] by an Analytical Schema (AnS), from which end-users can derive themselves many different cubes. However, there is still the need for a data administrator to

* This research is supported by ANR project PEGASE (ANR-16-CE23-0011).

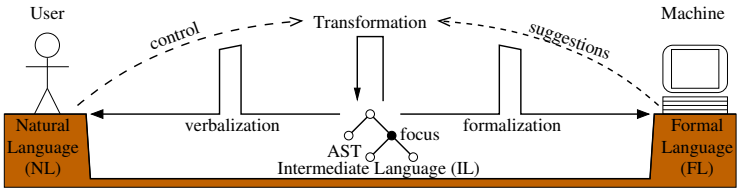


**Fig. 1.** Principle of the `N<A>F` design pattern

define the AnS. A consequence of the lack of direct access is a limited expressivity for the end-user, compared to the direct use of SPARQL 1.1 [10]. Indeed, every OLAP view can be expressed as a SPARQL aggregation query (where each SPARQL result corresponds to a cube cell), while each data cube allows for a limited range of questions. Each new question may require the definition of a new data cube, which can generally be done by the data administrator only. The counterpart of SPARQL's expressivity is that it is much more difficult for an end user to write SPARQL queries than to interact with an OLAP user interface. Another drawback is that SPARQL engines are not optimized for data analytics like OLAP engines. However, they are already usable in practice as we show in this work, their optimization is out of the scope of this paper.

In this paper, we propose an alternative approach that exploits the computation features of SPARQL 1.1 (aggregations, expressions) to directly answer analytical queries on *vanilla* RDF graphs, i.e. RDF graphs *not* customized to data analytics. We rely on the `N<A>F` design pattern [5] to design a query builder user interface that is *user-friendly* and *responsive*. In particular, it is user-friendly by completely hiding SPARQL behind a verbalization of the built queries in natural language. It is also made responsive by giving intermediate results and suggestions for refining the query, at each step, not only at the end of the building process. Therefore, the user interface of our approach shares features with QA (verbalization of queries and suggestions in natural language), and OLAP (interactive visualization of results).

The paper is organized as follows. Section 2 shortly explains and illustrates the `N<A>F` design pattern, which serves as a formalization framework for this work. Section 3 formalizes the building of analytical queries as a new instance of `N<A>F`. Section 4 presents an evaluation of the expressivity, responsiveness, and usability of our approach. A Web application[1] is available online, and includes the permalinks of about 70 analytical queries, and some screencasts.

## 2 The `N<A>F` Design Pattern

The purpose of the `N<A>F` design pattern [5] is to bridge the gap between natural languages (NL) and formal languages (FL, here SPARQL), as summarized in Figure 1. The user stands on the NL side, and does not understand the FL. The

[1] http://www.irisa.fr/LIS/ferre/sparklis/

machine, here a SPARQL endpoint, stands on the FL side, and does not understand the NL. The design pattern has already been instantiated to three tasks with different FLs [5]: (a) semantic search with SPARQL basic graph patterns (including cycles), and their combination with `UNION`, `OPTIONAL`, and `MINUS` (hereafter called *simple graph patterns* (SGP)) [7], (b) semantic authoring with RDF descriptions [8], and (c) ontology design and completion with OWL class expressions [6]. The central element of the bridge is made of the Abstract Syntax Trees (AST) of an Intermediate Language (IL), which is designed to make translations from ASTs to both NL (*verbalization*) and FL (*formalization*) as simple as possible. IL has no proper concrete syntax, NL and FL play this role, respectively for the user and the machine. ASTs are tree structures where the nodes represent query components and subqueries. Each node $X = \mathbf{C}(X_1, \ldots, X_n)$ is characterized by a *construct* $\mathbf{C}$, the type of the node, and a tuple of nodes $X_1, \ldots, X_n$, the children of $X$. `N<A>F` follows the query builder approach, where the structure that is incrementally built is precisely an AST. Unlike other query builders, the generated query (FL) and the displayed query (NL) may strongly differ in their structure thanks to the mediation of IL. The AST is initially the simplest query, and is incrementally built by applying *transformations*. A transformation may insert or delete a query component at the *focus*. The *focus* is a distinguished node of the AST that the user can freely move to control which parts of the query should be modified. Transformations are suggested by the machine based on query semantics and actual data, and controlled by users after they have been verbalized in NL.

We illustrate `N<A>F` on simple graph patterns (SGP). The AST

$$\mathbf{That}(\mathbf{A}(\texttt{:Film}), \mathbf{Has}(\texttt{:director}, \mathbf{Term}(\texttt{:Spielberg})))$$

is a nesting of constructs inspired by NL syntactic constructs, with RDF terms as atomic components (e.g., class `:Film`, term `:Spielberg`). The tree structure of the AST is traversed recursively to generate its translations to FL and NL.

- SPARQL: `?x a :Film. ?x :director :Spielberg.`
- English: `'a film whose director is Spielberg'`

The AST is built through the following sequence of transformations and ASTs (the current focus is underlined):

(0) initial AST : $\underline{\mathbf{Something}}$
(1) class `:Film` : $\underline{\mathbf{A}(\texttt{:Film})}$
(2) prop. `:director` : $\mathbf{That}(\mathbf{A}(\texttt{:Film}), \mathbf{Has}(\texttt{:director}, \underline{\mathbf{Something}}))$
(3) term `:Spielberg`: $\mathbf{That}(\mathbf{A}(\texttt{:Film}), \mathbf{Has}(\texttt{:director}, \underline{\mathbf{Term}(\texttt{:Spielberg})}))$
(4) move focus : $\mathbf{That}(\underline{\mathbf{A}(\texttt{:Film})}, \mathbf{Has}(\texttt{:director}, \mathbf{Term}(\texttt{:Spielberg})))$

The transformations are suggested according to the focus and actual results of the current query. For example, at step (2) only properties having a film as subject or object are suggested, and at step (3) only film directors are suggested as terms. At each step, the user interface shows: (a) the verbalization of the current query with the focus highlighted, (b) the results of the generated SPARQL query, and (c) the lists of suggested transformations.

## 3 Guided Building of Analytical Queries on RDF Graphs

The contribution of this paper is to fill the gap between the end-user and the computation features of SPARQL 1.1 (expressions, filters, bindings, and aggregations) to the purpose of the direct data analytics of vanilla RDF graphs. We realize this by designing a new instance of the `N<A>F` design pattern, i.e. new kinds of ASTs, along with their transformations, their formalization, and verbalization. Starting from the instance on simple graph patterns (see Section 2), this new instance introduces two new kinds of ASTs that cover the computation features of SPARQL, in addition to graph patterns: *tables* and *expressions*. The ASTs of analytical queries are *table* ASTs, and are composed of *graph pattern* ASTs and *expression* ASTs. A simple graph pattern AST $P$ evaluates to a set of mappings $M(P)$, where each mapping $\mu \in M(P)$ is a partial function from variables of the graph pattern to RDF terms. An expression AST $E$ evaluates to an RDF term, computed as $eval(E)$. A table AST $T$ evaluates to a tabular structure with a set of *columns* $C(T)$, and a set of *rows* $R(T)$ where each row $r \in R(T)$ is a partial mapping from columns to RDF terms.

In the following subsections we progressively cover the computation features of SPARQL by defining new AST constructs. Each subsection presents one or two use cases to motivate the new feature, and then formally defines the new construct. Use cases are based on a concrete dataset, Mondial [12], that contains geographical knowledge (e.g., countries, cities, continents, bodies of water) with a lot of numerical data (e.g., population, area).

The constructs of expression ASTs are defined at once in Section 3.2. For each new construct of a table AST $T$, we define:

- $C(T)$: the set of columns,
- $R(T)$: the set of rows (defined with relational algebra),
- $sparql(T)$: the formalization in SPARQL,
- $nl(T)$: the verbalization in NL (here English),
- a set of transformations to build the construct.

An important note is that function $sparql()$ is not always applied to the table AST itself but sometimes to a variation of it that depends on the focus. Variations are explained below where needed. The global formalization of a table $T$ is `SELECT * WHERE` { $sparql(T')$ } where $T'$ is the focus-dependent variant of $T$.

### 3.1 Primitive Tables

All of our computation constructs define a table as a function of another table, and so we need primitive tables to start with. Possible candidates for a primitive table are a table of a relational database, an OLAP data cube or a spreadsheet. However, in order to allow direct analytics of vanilla RDF graphs, we propose to use SPARQL simple graph patterns (SGP) to extract arbitrary tables of facts. Indeed, the results of a SPARQL `SELECT` query are returned as a table. We can then reuse previous work on the `N<A>F`-based building of SGPs, implemented in the Sparklis tool [7], to help users build those primitive tables.

**Definition 1.** *Let $P$ be a simple graph pattern AST, and $sparql(P)$ its translation to SPARQL. The table AST $T = \mathbf{GetAnswers}(P)$ represents the primitive table whose columns are the variables of $sparql(P)$, and rows are the solutions.*

- $C(T) = Vars(sparql(P))$
- $sparql(T) = sparql(P)$
- $R(T) = Sols(sparql(P))$
- $nl(T) =$ `'give me` $nl(P)$`'`

For example, if a user wants to build a primitive table of countries along with their population and area, four steps are enough to build the following AST:

$$T0 := \mathbf{GetAnswers}(\mathbf{That}(\mathbf{A}(\texttt{:Country}),\ \mathbf{And}(\mathbf{Has}(\texttt{:population}, \mathbf{Something}), \mathbf{Has}(\texttt{:area}, \mathbf{Something}))))$$

which translates to SPARQL: `?x1 a :Country. ?x1 :population ?x2. ?x1 :area ?x3.` and to English: `'give me every country that has a population and that has an area'`.

### 3.2 Expressions in Bindings and Filters

Use case (E): *Give me the population density for each country, from population and area.* SPARQL expressions (e.g., `?pop / ?area`) are used in two kinds of contexts: *filterings* (`FILTER`) and *bindings* (`SELECT`, `BIND`, `GROUP BY`). Given a table, a filtering performs a selection on the set of rows based on a Boolean expression. Given a table, a binding adds a column and computes its value for each row with an expression. In both cases, the expression can only refer to the columns of the table. In data analytics, filterings are important to select subsets of data, and bindings are important to derive information that is not explicitely represented, e.g. *population density* in use case (E).

We first define expression ASTs as they are a component of filterings and bindings. We define them in a classical way, as a composition of constants, variables, and operators/functions.

**Definition 2 (expression).** *An expression AST $E$ is composed of RDF terms (constants), table columns (variables), and SPARQL operators/functions. We add the* undefined expression *construct* ?? *to allow for the incremental building of expressions. An expression $E$ is said* defined *when it does not contain* ??*. We note $C(E)$ the set of columns refered to in the expression.*

The evaluation of an expression AST $E$ is denoted by $eval_r(E)$, where $r$ is the row on which the evaluation is performed. $eval_r(E)$ and $sparql(E)$ are defined in the obvious way, but are only defined when $E$ is defined itself. If the focus is on a subexpression $E'$, then only that subexpression is evaluated and formalized in SPARQL in order to show the value at focus. The verbalization of expression ASTs results from the nesting of the verbalization of its functions and operators. It mixes mathematical notations and text depending on which is the clearer: `'`$E_1$ `+` $E_2$`'` is clear to everybody and less verbose than `'the addition of` $E_1$ `and` $E_2$`'`, while `'`$E_1$ `or` $E_2$`'` is less obscure than `'`$E_1$ `||` $E_2$`'` for non-IT people.

The verbalization of columns is derived from the names of classes and properties used in the graph pattern that introduce them as variables: e.g., in $sparql(T0)$, `'the population'` for $x_2$, `'the area'` for $x_3$. The parts of the expression that are not under focus are displayed in a different way (e.g., gray font) to show that they are not actually computed.

The transformations used to build expression ASTs are the following: (a) insert an RDF term at focus, (b) insert a column at focus, (c) apply an operator or a function at focus. When an operator/function is applied, the focus is automatically moved to the next undefined expression if any (e.g., other function arguments), and to the whole expression otherwise. For instance, the sequence of ASTs and transformations that builds the expression of use case (E), which computes population density from population and area, is the following (the focus is underlined):

(0) inital expression : $E = \underline{??}$
(1) insert column $p$ (population): $E = \underline{p}$
(2) apply operator / (division) : $E = p \ / \ \underline{??}$
(3) insert column $a$ (area) : $E = \underline{p \ / \ a}$

There are constraints on which transformations are applicable so as to avoid misformed expressions. Only columns that are in scope can be inserted. That scope is defined by table AST constructs that contain expressions. Type constraints are also used to determine which operators and functions can be applied, and for which datatypes RDF terms can be inserted. In previous example, at step (2) only numeric operators/functions can be applied because $p$ is an integer, and at step (3) only numeric columns and terms can be inserted.

We now define the two table constructs that contain an expression.

**Definition 3 (filtering).** *Let $T_1$ be a table AST, and $E$ be a Boolean expression AST s.t. $C(E) \subseteq C(T_1)$. The table AST $T =$ **SelectRows**$(T_1, E)$ represents a filtering of the rows of $T_1$ that verify $E$.*

- $C(T) = C(T_1)$
- $R(T) = \begin{cases} \{r \mid r \in R(T_1), eval_r(E) = true\} & \text{if } E \text{ is defined,} \\ R(T_1) & \text{otherwise} \end{cases}$
- $sparql(T) = \begin{cases} sparql(T_1) \texttt{ FILTER ( } sparql(E) \texttt{ )} & \text{if } E \text{ is defined} \\ sparql(T_1) & \text{otherwise} \end{cases}$
- $nl(T) =$ `'`$nl(T_1)$ `where` $nl(E)$`'`

**Definition 4 (binding).** *Let $T_1$ be a table AST, $x$ be a column s.t. $x \notin C(T_1)$, and $E$ be an expression AST s.t. $C(E) \subseteq C(T_1)$. The table AST $T =$ **AddColumn**$(T_1, x, E)$ represents the addition of a column $x$ to $T_1$, and the binding of $x$ to $E$.*

- $C(T) = C(T_1) \cup \{x\}$
- $R(T) = \begin{cases} \{r \cup \{x \mapsto eval_r(E)\} \mid r \in R(T_1)\} & \text{if } E \text{ is defined,} \\ R(T_1) & \text{otherwise } (x \text{ is unbound}) \end{cases}$

– $sparql(T) = \begin{cases} sparql(T_1)\ \texttt{BIND (}\ sparql(E)\ \texttt{AS ?}x\ \texttt{)} & \textit{if } E \textit{ is defined,} \\ sparql(T_1) & \textit{otherwise} \end{cases}$

– $nl(T) = \begin{cases} \texttt{`}nl(T_1)\ \texttt{and give me}\ name(x)\ \texttt{=}\ nl(E)\texttt{'} & \textit{if } name(x) \textit{ is defined,} \\ \texttt{`}nl(T_1)\ \texttt{and give me}\ nl(E)\texttt{'} & \textit{otherwise} \end{cases}$

In both constructs, the columns that are in scope of the expression are the columns $C(T_1)$. If the focus is on a subexpression $E'$, then function $sparql()$ is applied respectively to $T' = \mathbf{SelectRows}(T_1, E')$ and $T' = \mathbf{AddColumn}(T_1, x, E')$, thus ignoring the rest of the expression in the computation. It suffices to move the focus upward in the AST in order to recover the complete computation.

Filterings and bindings are introduced in the AST by putting the focus on a column, and by applying a function or operator, this initiates the building of an expression. The choice between **SelectRows** and **AddColumn** is based on the type of the whole expression under the assumption that Boolean expressions are primarily used to select rows. However, another transformation allows to force the choice of **AddColumn** to allow a column of Boolean values. Finally, there is a transformation to give a user-defined name $name(x)$ to the new column, which can be used in the verbalization. For instance, starting from the primitive table $T0$ defined in Section 3.1, use case (E) can be built through the following sequence of transformations and ASTs:

(0-3) ... : $T = T0$ (see Section 3.1)
(4) focus on $x_2$ (population):
(5) apply operator / : $T = \mathbf{AddColumn}(T0, x_4, x_2\ /\ \underline{??})$
(6) insert column $x_3$ (area) : $T = \mathbf{AddColumn}(T0, x_4, \underline{x_2\ /\ x_3})$
(7) name column $x_4$ as `'population density'`:

The resulting table AST can be translated to SPARQL:

```
?x1 a :Country. ?x1 :population ?x2. ?x1 :area ?x3. BIND (?x2/?x3 AS ?x4)
```

and to English:

```
'give me every country that has a population and that has an area,
 and give me the population density = the population / the area'.
```

### 3.3 Aggregations

We call a *basic aggregation* the application of an aggregation operator on a set of entities or values, resulting in a single value. Use case (A1): *How many countries are there in Europe?*. A *simple aggregation* consists in making groups out of a set of values according to one or several criteria, and then applying an aggregation operator on each group of values. A *multiple aggregation* extends simple aggregation by having several aggregated values for each group. Use case (A2): *Give me the average population and the average area of countries, for each continent.* An aggregation corresponds to an OLAP view, where the grouping criteria are its *dimensions*, and where the aggregated values are its *measures*. In SPARQL, aggregations rely on the use of aggregation operators in the `SELECT` clause, and

on `GROUP BY` clauses. We introduce a new construct for table ASTs that cover all those aggregations.

**Definition 5 (aggregations).** *Let $T_1$ be a table AST, $X \subseteq C(T_1)$ be a set of columns (possibly empty), and $G = \{(g_j, y_j, z_j)\}_{j \in 1..m}$ be a set of triples (aggregation operator, column, column) s.t. for all $j \in 1..m$, $y_j \in C(T_1) \setminus X$ and $z_j \notin C(T_1)$. The table AST $T = Aggregate(T_1, X, G)$ represents the table obtained by grouping rows in $T_1$ by columns $X$, and for each $(g_j, y_j, z_j) \in G$, by binding $z_j$ to the application of $g_j$ to the multiset of values of $y_j$ in each group.*

- $C(T) = X \cup \{z_j\}_{j \in 1.m}$
- $R(T) = \{r_X \cup \{z_j \mapsto g_j(V_j)\}_{j \in 1..m} \mid r_X \in \pi_X R(T_1)$, *for each* $(g_j, y_j, z_j) \in G$, $V_j = \{\{r(y_j) \mid r \in R(T_1), \pi_X r = r_X\}\}$ $\}$
- $sparql(T) =$ { `SELECT ?`$x_1 \ldots$`?`$x_n$ `(`$g_1$`(?`$y_1$`) AS ?`$z_1$`)` $\ldots$ `(`$g_m$`(?`$y_m$`) AS ?`$z_m$`)` `WHERE {` $sparql(T_1)$ `} GROUP BY ?`$x_1 \ldots$`?`$x_n$ }
- $nl(T) =$ ‘$nl(T_1)$ `and [for ...each` $nl(x_i)$`,...] give me ...`$nl(z_j)$`...`’

In $C(T)$, the $y$-columns disappear and are replaced by the aggregated $z$-columns. In the definition of $R(T)$, the notation $\{\{...\}\}$ is for multisets (a.k.a. bags), and the distinction with sets is important for aggregators such `AVG` or `SUM`. Because other table constructs generate a SPARQL graph pattern, we here use a subquery in the definition of $sparql(T)$ to allow the free combinations of different kinds of computations (see Section 3.4 for examples). If the focus is in $T_1$ then both $R()$ and $sparql()$ are applied to $T_1$ instead of $T$, hence ignoring the aggregation. In this way, the columns that are hidden by the aggregation ($C(T_1) \setminus X$) can be temporarily accessed by moving the focus in $T_1$. The verbalization of each aggregated column $z_j$ is the verbalization of $g_j(y_j)$: e.g., ‘`the number of` $nl(y_j)$’, ‘`the average` $nl(y_j)$’. The brackets around ‘`for each...`’ indicate an optional part, in the case where $X = \emptyset$.

Similarly to bindings and filters, an aggregation is introduced in the AST by moving the focus on a column, and by applying an aggregation operator. Then, other columns can be selected, either as grouping criteria or as an additional aggregated column. Type constraints are also used here to restrict which aggregator can be applied to which column. Here is the sequence of transformations that leads to use case (A2) from primitive table $T0$:

(0-4) ... : $T = T0$ (see Section 3.1)
(5) property `:continent` : $T = T1 =$ **GetAnswers**$(\ldots,$ **Has**$($`:continent`$, \ldots))$
(6) move focus on $x_2$ (population)
(7) apply aggregation `AVG` : $T =$ **Aggregate**$(T1, \{\}, \{(\texttt{AVG}, x_2, x_5)\})$
(8) group by $x_4$ (continent): $T =$ **Aggregate**$(T1, \{x_4\}, \{(\texttt{AVG}, x_2, x_5)\})$
(9) apply `AVG` on $x_3$ (area) : $T =$ **Aggregate**$(T1, \{x_4\}, \{(\texttt{AVG}, x_2, x_5), (\texttt{AVG}, x_3, x_6)\})$

Different sequences are possible, elements can be introduced in almost any order, and it is also possible to remove query parts. For example, in (A2), it is possible to first build the query without column $x_4$ in $T1$ (continent), and without grouping by $x_4$. This computes the average population and area over all countries. Then the focus can be moved back in $T1$ on $x_1$ (country), insert

property `:continent` to add column $x_4$ in $T1$, and finally move the focus back on the aggregation to group by continent. The resulting AST for (A2) can be translated to SPARQL:

```
{ SELECT ?x4 (AVG(?x2) AS ?x5) (AVG(?x3) AS ?x6)
   WHERE { ?x1 a :Country. ?x1 :population ?x2.
           ?x1 :area ?x3. ?x1 :continent ?x4. }
   GROUP BY ?x4 }
```

and to English:

```
'give me every country
    that has a population and that has an area and that has a continent
and for each continent,
    give me the average population and the average area'
```

### 3.4 Combinatorics of Table Constructs

The real power of our approach lies in the combinatorics of the above table constructs (**GetAnswers**, **SelectRows**, **AddColumn**, **Aggregate**), which can be chained arbitrarily to build more and more sophisticated tables. We illustrate this with three new use cases.

**Aggregation of bindings.** Use case (C1): *What is the average GDP per capita, for each continent?*. This use case requires to retrieve information about countries, to compute the GDP per capita as (total GDP $\times 10^6$/ population) for each country, and to average the GDP per capita over each continent. The query can be built in 11 steps, producing 3 nested table constructs.

**Comparison of aggregations.** Use case (C2): *Which continents have an average agricultural GDP greater than their average service GDP?*. This use case requires to retrieve information about countries, to compute two aggregations for the same grouping (multiple aggregation), and then to express an inequality between the two aggregations (filtering). It can be built in 9 steps, and a nesting of 3 table constructs.

**Nested aggregations.** Use case (C3): *Give me for every number of islands in an archipelago, the number of archipelagos having that number of islands.* This use case requires to retrieve islands and their archipelagos, to compute the number of islands per archipelago (simple aggregation), and then to compute for each number of islands, the respective number of archipelagos (simple aggregation). The query can be built in 6 steps, and 3 nested table constructs.

### 3.5 Implementation

We have fully implemented our approach into Sparklis. Its previous version covered all simple graph patterns, and therefore provided everything needed for the building of our primitive tables. Thanks to the genericity of `N<A>F`, no change was required in the user interface. The impact of our approach appears to users only through a richer query language, and additional suggestions. This makes it easy for users to transit to the new version. On the implementation side, however, a major refactoring of the intermediate language was necessary with the

introduction of the new kinds of ASTs for tables and expressions in addition to simple graph patterns. Entirely new components were also introduced, e.g., type inference and type checking for computing some of the new suggestions.

## 4 Evaluation

We conducted two evaluations. The first evaluates expressivity and responsiveness from our participation to the QALD-6 challenge. The second evaluates usability with a user study comparing Sparklis to a SPARQL editor.

### 4.1 Evaluation on the QALD-6 Challenge

The QALD-6 challenge (Question Answering over Linked Data) introduced a new task on "Statistical question answering over RDF datacubes" [16]. The dataset contains about 4 million transactions on government spendings all over the world, organized into 50 datacubes. There are about 16M triples in total. Note that, although this dataset is represented as a set of datacubes, we answered the challenge questions by building primitive tables from graph patterns, like we would do for any other RDF dataset. We officially took part in the challenge and submitted the answers that we obtained from Sparklis by building queries.

**Expressivity.** We evaluated expressivity by measuring the coverage of QALD-6 questions. Out of 150 questions (training+test), 148 questions are basic or simple aggregations, and are therefore covered by our approach; and 2 questions (training Q23 and test Q23) are comparisons of two aggregations, and are not covered by our approach. The reason is that the two aggregations use disconnected graph patterns, whereas our approach is limited to a single connected graph pattern. In the challenge, we managed to answer 49/50 test questions, out of which 47 were correct, hence a success rate of 94% (official measure: $F_1 = 0.95$). The two errors come from an ambiguity in questions Q35 and Q42, which admit several equally plausible answers (URIs with the same label).

**Responsiveness.** We evaluated responsiveness by measuring the time needed to build queries in our implementation by a user mastering its user interface (the author of this paper). The measures include user interactions and are therefore an upper bound on the system runtime. The query building time ranged from 31s to 6min20s, and half of the queries were built in less than 1min30s (median time). Most QALD-6 questions need 5-10 steps to build the query. It shows that our implementation is responsive enough to satisfy real information needs on large datasets.

### 4.2 Comparison with Writing SPARQL Queries in Yasgui

**Methodology.** The objective of this user study is to evaluate the benefits of our approach compared to writing SPARQL queries directly. The subjects were 12 pairs of post-graduate students in Computer Science applied to Business Management. They all recently attended a Semantic Web course with about 10h of

teaching and practice of SPARQL. Their task was to answer two similar series of 10 questions on the MONDIAL dataset, covering all kinds of SPARQL computations. The subjects had to use a different system for each series of questions: Yasgui [14], a SPARQL query editor improved with syntax highlighting, and our Sparklis-based implementation. The subjects had only a short presentation of Sparklis before, and no practice of it. Each subject was randomly assigned an order (Yasgui first vs Sparklis first) to avoid bias. For each system/series, they were given 10min for setup, and 45min for question answering. To help the writing of SPARQL queries, they were given the list of classes and properties used in MONDIAL. Finally, the subjects filled in a questionnaire to report their feelings and comments.

**Objective results.** With Yasgui, only the first question was correctly answered by the majority, instead of 5 questions (out of 10) for Sparklis. On average, the number of subjects who correctly answered a question was 3.8 times higher for Sparklis compared to writing SPARQL queries. With Yasgui, the subjects managed to produce answers for 1-3 questions, 1.67 on average. The rate of correct answers is 71%. The best subject answered 3 questions, all correctly, and with an average of 15min per question. In comparison, with Sparklis, the subjects managed to produce answers for 3-10 questions, 6.17 on average. This is 3.7 more questions, and the weakest result for Sparklis is equal to the strongest result for Yasgui. The rate of correct answers is also higher at 85%. Most errors (8/13) were done by only two subjects out of 12. The most common error is the omission of groupings in aggregations, suggesting to make them more visible in the user interface. The best subject answered 10 questions, 8 of which correct, and with an average of 4.5min per question. The time spent per question is generally higher for the first 2 questions, and generally stays under 10min for the other questions, although they are more complex, and can be as low as 2-3min. Those results demonstrate that, even with practice of SPARQL query writing and no practice of Sparklis, subjects quickly learn to use Sparklis, and are much more effective with it. The weakest subject on Sparklis is still as good as the strongest subject on Yasgui. Moreover, all subjects had better results with Sparklis than with Yasgui.

**Subjective results.** Over the 12 subjects, 9 *clearly prefer Sparklis* and 3 *would rather use Sparklis*, hence unanimous preference for Sparklis. On a 0-10 scale, Yasgui got marks between 2 and 8, 4.8 on average, and Sparklis got marks between 7 and 10, 8.3 on average (half of the subjects gave 9-10 marks).

## 5 Conclusion

We have shown how SPARQL 1.1 can be leveraged to offer rich data analytics on vanilla RDF graphs through a guided query builder user interface. We have implemented the approach in Sparklis, and validated its expressivity, responsiveness, and usability through the QALD-6 challenge and a user study. The extended Sparklis has been officially adopted as a query tool by Persée[2],

[2] http://data.persee.fr/

a French organization that provides open access to more than 600,000 scientific publications to researchers in humanities and social sciences.

## References


1. Atzori, M., Mazzeo, G., Zaniolo, C.: QA$^3$: a natural language approach to statistical question answering (2016), http://www.semantic-web-journal.net/system/files/swj1847.pdf, submitted to the Semantic Web journal
2. Chaudhuri, S., Dayal, U.: An overview of data warehousing and OLAP technology. ACM Sigmod record 26(1), 65–74 (1997)
3. Colazzo, D., Goasdoué, F., Manolescu, I., Roatiş, A.: RDF analytics: lenses over semantic graphs. In: Int. Conf. World Wide Web. pp. 467–478. ACM (2014)
4. Cyganiak, R., Reynolds, D., Tennison, J.: The RDF data cube vocabulary (2013)
5. Ferré, S.: Bridging the gap between formal languages and natural languages with zippers. In: Sack, H., et al. (eds.) Extended Semantic Web Conf. (ESWC). pp. 269–284. Springer (2016)
6. Ferré, S.: Semantic authoring of ontologies by exploration and elimination of possible worlds. In: Int. Conf. Knowledge Engineering and Knowledge Management. LNAI 10024, Springer (2016)
7. Ferré, S.: Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. Semantic Web: Interoperability, Usability, Applicability 8(3), 405–418 (2017), http://www.irisa.fr/LIS/ferre/sparklis/
8. Hermann, A., Ferré, S., Ducassé, M.: An interactive guidance process supporting consistent updates of RDFS graphs. In: ten Teije, A., et al. (eds.) Int. Conf. Knowledge Engineering and Knowledge Management (EKAW). pp. 185–199. LNAI 7603, Springer (2012)
9. Höffner, K., Lehmann, J., Usbeck, R.: CubeQA - question answering on RDF data cubes. In: Int. Semantic Web Conf. pp. 325–340. Springer (2016)
10. Kaminski, M., Kostylev, E.V., Cuenca Grau, B.: Semantics and expressive power of subqueries and aggregates in SPARQL 1.1. In: Int. Conf. World Wide Web. pp. 227–238. ACM (2016)
11. Kovacic, I., Schuetz, C.G., Schausberger, S., Sumereder, R., Schrefl, M.: Guided query composition with semantic OLAP patterns. In: EDBT/ICDT Workshops. pp. 67–74 (2018)
12. May, W.: Information extraction and integration with Florid: The Mondial case study. Tech. Rep. 131, Universität Freiburg, Institut für Informatik (1999), available from http://dbis.informatik.uni-goettingen.de/Mondial
13. Papadaki, M.E., Tzitzikas, Y., Spyratos, N.: Analytics over RDF graphs. In: Int. Work. Information Search, Integration, and Personalization. pp. 37–52. Springer (2019)
14. Rietveld, L., Hoekstra, R.: YASGUI: Not just another SPARQL client. In: The Semantic Web: ESWC 2013 Satellite Events, pp. 78–86. Springer (2013)
15. Sherkhonov, E., Grau, B.C., Kharlamov, E., Kostylev, E.V.: Semantic faceted search with aggregation and recursion. In: d'Amato, C., et al. (eds.) Int. Semantic Web Conf. (ISWC). pp. 594–610. LNCS 10587, Springer (2017)
16. Unger, C., Ngomo, A.C.N., Cabrio, E.: 6th open challenge on question answering over linked data (QALD-6). In: Sack, H., et al. (eds.) Semantic Web Evaluation Challenge. pp. 171–177. Springer (2016)