Open Data Analytic Querying using a Relation-Free API

Lucas F. de Oliveira, Alessandro Elias, Fabiola Santore, Diego Pasqualin, Luis C. E. Bona, Marcos Sunyé and Marcos Didonet Del Fabro

C3SL Labs, Informatics Department, Federal University of Paraná, Curitiba, Brazil

Keywords: Analytical Querying, Open Data API, Query Generation, Relation-Free Query.

Abstract: The large availability of tabular *Open Data* sources with hundreds of attributes and relations makes the query development a difficult task, where analytic queries are common. When writing such queries, often called SPJG (Select-Project-Join-GroupBy), it is necessary to understand a data model and to write JOIN operations. The most common approach is to use business intelligence frameworks, or recent solutions based on keywords or examples. However, they require the utilization of specific applications and there is a lack of support for web-based APIs. We present a solution that eases the task of query development for tabular *Open Data* analytics through an API, using a simplified query representation where it is not allowed to specify the data relations, and consequently neither the joins over them, called *Relation-Free Query*. We define a single virtual schema that captures the database structure, which allows the use of relation-free queries in existent DBMS's. The concrete queries are exposed by a RESTful API, which is then translated into a database query language using known query generation solutions. The API is available as a microservice. We present a case study to describe solution, using a real world scenario to query in an integrated database of several Brazilian open databases with hundreds of attributes.

1 INTRODUCTION

The amount of data available in *Open Data* sources has not stopped growing in recent years. The need to explore and correlate these data has given rise to new roles such as data scientists, who even though are not database experts, they have the task of joining and processing a large amount of different data sources. There are several Open Data formats, which may be semi-structured (such as CSVs) or unstructured ones (such as JSON documents or free text). We focus on *semi-structured data* that can be extracted and integrated into relational databases. The enhancement of data integration techniques for Open Data integration (Miller, 2018) provides means to produce large queries with several relations and attributes.

In order to retrieve data from a relational database, a typical query has four elements: (1) the attributes to be retrieved, (2) the restrictions to be satisfied, (3) the relations that are used and (4) how to combine these relations. These queries are often called as *SPJ* (Select-Project-Join) queries. When adding grouping, we can call them *SPJG* (Select-Project-Join-GroupBy) queries. The relations to use and how to combine then are dependent of the database.

We consider that in the Open Data and data scientists scenario, it is often desired to perform simple analytic queries, also known as '*stupid analytic*' (Abadi and Stonebraker, 2015), with a *SPJG* format. In addition, it is important to have the data accessible though an API, so it could be consumed by application developers, broadening the access of available public data.

We categorize existing solutions into three groups. First, the utilization of Business Intelligence frameworks, such as *Saiku*, *IBM Cognos* or *QLikView*, or many others. They provide complete frameworks accessible for the developers or data analysts, often providing graphical interfaces or specialized languages, such as MDX (multidimensional Expressions) to help on query design, using the dimensional model. Second, there are solutions based on keywords, examples or simplified SQL-like languages. The *Cheetah* framework (Chen, 2010) presents the notion of virtual tables, where some query design aspects are hidden. The SODA system (Blunschi et al., 2012), SQAK (Tata and Lohman, 2008) or the work from (Bergamaschi et al., 2011) ease query design using keywords. These approaches rely on prior knowledge about the instances or about the schema to produce a set of possible queries. Finally, the solutions focusing on producing accessible APIs, such as (Ed-douibi et al., 2018) or (Sellami et al., 2014).

F. de Oliveira, L., Elias, A., Santore, F., Pasqualin, D., Bona, L., Sunyé, M. and Didonet Del Fabro, M.
Open Data Analytic Querying using a Relation-Free API.
DOI: 10.5220/0009413001480155
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 1*, pages 148-155
ISBN: 978-989-758-423-7

In this paper, we present an approach for *SPJG* query development using an API. We call our query representation format *RFQ* (Relation-Free-Query), where we do not write the relations and the joins operations, focusing on what to return (aggregated and categorized measures, as well as additional attributes). This format is possible because we provide a global *virtual schema* that represents the integrated information. We also present how to translate such representation into *SQL* queries, by applying known query generation techniques. We define a *RESTful API* as the top layer to write queries. Thus, the solution is made available by an Open Data microservice, which is currently the most common access method chosen by application developers.

We present a case study contrasting a *RFQ* query with its corresponding generated *SQL* code and API call. The database used consists of an integrated database from two initially separated data sources, with different degrees of normalization. This database contains 24 relations, more than 700 attributes, and about 2.5 billions of records.

The integrated sources, called *BIOD (Blended Integrated Open Data)*, are publicly avaiable as a microservice [1]. The framework is called called *BlenDb* [2] provided as a *Free Software* under AGPL Licence.

This paper is organized as follows: section 2 presents the relation free query definitions; section 3 shows how to translate relation free queries to *SPJG* queries, and the RESTful API format; section 4 describe our case study; section 5 contains the related work and section 6 the conclusions.

2 RELATION-FREE QUERY

In this section we present our approach to design relation free queries (*RFQ*) over relational databases, its central definitions and a driving example that is used in the remaining of the paper.

2.1 Definitions

To answer *SPJG* queries, filters are used to define constraints, which the returned records must satisfy. We allow the specification of aggregation and grouping. The attributes that are aggregated are called metrics, and the attributes that are grouped, dimensions. These metrics and dimensions are used to project the attributes in the tuples.

[1]BIOD microservice: https://biod.c3sl.ufpr.br/index_en.html

[2]BlenDb source: https://gitlab.c3sl.ufpr.br/c3sl/blendb

Definition 1 (Attribute)**.** *An attribute a is the definition of the name and datatype of collections of values.*

Definition 2 (Dimension)**.** *A dimension d is an attribute a, which will be used to perform grouping.*

In a relational database, an attribute corresponds to the *column* definition of a *table*. The dimensions are used to categorize and to define degree of detail (granularity) of the data.

Definition 3 (Metric)**.** *A metric m is a pair* $< f, a >$*, where f is an aggregation function and a is an attribute, where:*

- *f is one of 5 different functions:* ***SUM****,* ***AVG****,* ***MAX****,* ***MIN****,* ***COUNT****;*
- *a is the attribute that is aggregated.*

The metrics are the information that are aggregated, with a given function. The different kinds of functions are typical aggregation functions supported by existing DBMS.

Definition 4 (Filter)**.** *A filter f is a triple* $< d, o, v >$*, where d is a dimension, o is an operator and v is a value, and:*

- *the operator o is a binary operator, with the following possible values:* $>$, $<$, $\geq$, $\leq$, $=$ *or* $\neq$.
- *the value v is a constant, comparable with d.*

Definition 5 (Clause)**.** *A clause c is a set of filters* $\{f_0, ..., f_n\}$.

A filter represents a restriction over a dimension. The filters are grouped into clauses written in *Conjunctive Normal Form* (CNF). Filters in the same clause are combined using the OR operator and the clauses are combined using the AND operator.

The above definitions is similar to existing multidimensional data model definitions (e.g. (Aligon et al., 2014)), but they have restrictions, since, for instance, we do not present the notion of a cube nor make assumptions about dimensions hierarchies.

Definition 6 (Relation Free Query)**.** *A RFQ Q is a triple* (M, D, C) *where M is a set of metrics* $\{m_0, ..., m_n\}$*, D is a set of dimensions* $\{d_0, ..., d_p\}$ *and C is a set of clauses* $\{c_0, ..., c_q\}$.

A *RFQ* can be denoted using the format below:

$$Q(m_0, m_1, ..., m_n)(d_0, d_1, ..., d_p)(c_0, c_1, ..., c_q)$$

The set of parenthesis contains the elements of the M, D and C sets respectively. This notation is inspired by *conjunctive query* notation (Abiteboul et al., 1995), with simplifications to remove the relations and to support only filters, metrics and dimensions. Note that in this section only the model is presented, it does not have a concrete syntax which allows use a existent DBMS. How to write concrete and executable queries is described later.

2.2 Driving Example

Consider a database with information about all educational institutions in Brazil. Such database is a simplified version of the database used in our case study. Consider the relations below:

- $student(st\_name, st\_id, sc\_id, grade, age)$
- $school(sc\_name, sc\_id, city\_id, category)$
- $city(city\_id, city\_name, state, region)$

The database contains where each student studies, the type of school and its geo-location data. We describe the *RFQ* and a corresponding *SQL* query.

Consider the following question: "What is the mean age and how many students of the forth grade exist by region?".

We define a metric number of students: $n\_student = (COUNT, st\_id)$, which uses the function $COUNT$ in the attribute st_id. We also define the metric mean age, $mean\_age = (AVG, age)$, as the function AVG over the attribute age. The question previously mentioned can be written in *RFQ* as:

$$Q(n\_student, mean\_age)(region)(\{grade = 4\})$$

The corresponding *SQL* query is:

```
SELECT   COUNT(st.st_id) AS n_student,
   AVG(st.age) AS mean_age, c.region
FROM student st
   INNER JOIN school sc ON sc.sc_id = st.sc_id
   INNER JOIN city c ON sc.city_id = c.city_id
WHERE st.grade = 4
GROUP BY c.region
```

When using *RFQ*, we only list the attributes and constraints. We highlight the most important similarities and differences between the two queries. First, the set of attributes returned in the *SQL* query matches exactly with the set of metrics and dimensions in the *RFQ*. The set of constraints (WHERE statement) also matches, and the attributes used in the constraints are not required to be in the attributes returned. All the attributes in the dimension set are in the GROUP BY statement and the all metrics have a corresponding aggregation function.

The way to calculate *n_student* and *mean_age*, which relations are used and how to combine then (Join operations) are not explicit in *RFQ*. The complete query is produced in the translation process of *RFQ* into a *SQL* query.

3 RFQ TO SQL

Relation-free queries are an alternative higher level representation to query over a relational database. This means they need to be translated into *SQL* queries.

The translation of a *RFQ* into relational algebra expression and then to a *SQL* query is done in two steps. First, the *RFQ* is translated into relational algebra and then into a valid *SQL* query over a virtual database schema. The second step converts the *SQL* over the virtual schema into a *SQL* in the real database schema. We detail these steps in the following sections.

3.1 The Virtual Schema

A *RFQ* query does not contain what relations are used and how to rename the attributes, for instance, that $n\_student = (COUNT, st\_id)$. This lack of information creates a "gap" in the query, which does not allow its direct translation into SQL.

In the first step of the translation, we define a virtual database schema to "fill the gap" in the query. The virtual schema is an abstract database schema used to transform a query in relation-free format into a query in *SQL* syntax. The virtual schema is built in function of the set of metrics, the set of dimensions and the real database structure. The intent of the virtual schema is to simulate a database that contains only one relation, containing all the attributes. Such restriction in the database schema explains why *RFQ* are more compact than *SQL* queries. If a database contains only one relation and self-joins are not allowed, the reference to the relation in **all queries** is the same, so, it is redundant in the queries. This is the principle that allows *RFQ* to omit relations.

However, relational databases with a single relation are rare. To overcome this issue we use query rewriting algorithms, to convert queries between different database schemes.

The virtual schema is formed by relations that represent views over the real database schema. The intent of the virtual schema was to be a schema with only one relation, however, because aggregation and grouping are allowed, the virtual schema must contain one relation per metric, similar to a star schema. To preserve the ability of omitting the relations, we impose restrictions over the schema and queries. These restrictions may not allow *RFQ* to represent all *SPJG* queries. However, it fits to simple analytics applications.

Definition 7 (Virtual Schema)**.** *A virtual schema* $V = (R, M, D)$ *is a set of views generated from a real schema R and a set of metrics M and a set of dimensions D where, for* $m \in M$ *exists a single view* A_m*. The relation* A_m *contains m and all dimensions that m can be grouped by.*

By the virtual schema definition we extract three properties. First, m is in exactly one view, denoted A_m. Second, all queries involving m must use A_m. Third, as A_m contains all dimensions that m can be grouped by, only A_m is required to aggregate a single metric m.

There are no restrictions in how to generate A_m from R, however, some operations can corrupt the semantic value of a metric. For instance, a join of a "n-to-n" relationship could replicate some records, which may duplicate *COUNT* operations. In our approach, used in the study case, we generate the A_m relations only through INNER JOINS, which avoids such issue.

3.2 Relation-Free Query to SQL

This section defines the translation process from *RFQ* to relational algebra expression and then to *SQL* queries. Given a *RFQ* $Q(M,D,C)$, we want to produce a translation of Q denoted as Q'. First we describe the set of relational algebra expressions to perform the translation then a brief explanation of the expression.

To keep the relational algebra expression simple we make a set of assumptions. In the filter operator, if the constraint requires an attribute that is not contained in the relation, this constraint is ignored. If the projection or grouping operation requires an attribute that is not contained in the relation, this attribute is ignored as well. The expression could be rewritten without these assumptions, however, we consider that the expression becomes oversized and more difficult to understand.

Let $m \in M$, and γ_G be the aggregation/grouping operation of the extended relational algebra. The query Q' over the virtual schema is defined as:

1. $Q'_m = \gamma_{D,f(m)}(\sigma_C(A_m))$
2. $Q' = Q'_{m_1} \bowtie Q'_{m_2} \bowtie ... \bowtie Q'_{m_n}$

The equation 1 is a sub-query, denoted $Q'm$ which defines the translation of a single metric query. The Q'_m is the relational algebra expression equivalent of the *RFQ* $Q_m(m)(D)(C)$. First we apply the filtering operator in the A_m relation, removing all tuples which do not satisfy the constraints in C. Then the filtered relation is aggregated, the metric m is aggregated using function f and grouped by D.

In equation 2 we define Q' as the INNER JOIN of all Q'_m. There is a Q_m sub-query for each metric $m \in M$. Equation 2 ensures that the query contains all required metrics. In other words, the Q' query aggregates each metric separately than joins the aggregated results.

Assuming that the relation A_m exists in the virtual schema, for all metrics in M, a *RFQ* can be converted to *SQL* over the virtual schema. The generated queries always follow the *SPJG* format, due to domain specific nature of RFQ for analytic queries. The joins are always INNER joins. Note that at this point it is not necessary to have a SQL query, which is produced in the next step.

We use the illustrative example to describe the same query, now over the virtual schema. It uses two A relations: $A_{n\_student}$ and A_{mean_age}.

$A_{n\_student}$ and A_{mean_age} are defined as the join of all relations (*student*, *school* and *city*) with one additional attribute for the metric. In $A_{n\_student}$ the additional attribute is *n_student* with is a copy of the attribute *st_id*. In A_{mean_age} the additional attribute is *mean_age* with is a copy of the attribute *age*. With this virtual schema, we can translate the *RFQ* used in the first example into the expression below.

$$Q'_{n\_student} = \gamma_{region,COUNT(n\_student)}(\sigma_{\{grade=4\}}(A_{n\_student}))$$
$$Q'_{mean\_age} = \gamma_{region,AVG(mean\_age)}(\sigma_{\{grade=4\}}(A_{mean\_age}))$$
$$Q' = Q'_{n\_student} \bowtie Q'_{mean\_age}$$

This expression is further translated into a SQL over the virtual schema.

3.3 From Virtual to a Concrete Schema

With Q' defined as a *SQL* query, the only step required is to adapt the query in the virtual schema to the real schema. This virtual schema is defined as a set of views over the real database schema. To make a *SQL* executable in a real schema, we use a query rewriting algorithm that receives the query Q' and views definitions of the virtual schema. As the relations in the virtual schema are defined as views in the real schema, the rewriting algorithm is an expansion in the query Q' of all the occurrences of the view v by its definition (similar to the ones used in GAV-approaches (Lenzerini, 2002) (Kwakye et al., 2013)).

After expanding all view references, the new query refers only to relations in the real databases and it can be executed. In other words, to make the database compatible with *RFQ*, the central requirement is the creation of the virtual schema respecting the constraints presented.

3.4 Creating RFQs using a RESTful API

In order to use *RFQ*, we implemented the **BlenDb tool**, which is a middleware that works as query generator between the input request and the target DBMS, i.e., it publishes an Open Data microservice that receives an *RESTful API* (Richardson et al., 2013) request and translates it into SQL. We choose an *RESTful API* because it is a simple format which is largely used to access *Open Data* sources though the Web, in virtually all kinds of application scenarios. This ubiquity could ease the adoption of the solution, instead of producing a new language from scratch. In addition, for a given service, it is not necessary to install or do any additional configuration by application developers.

The tool translates the requests to *SQL* and then makes a call to the DBMS to perform the query. When the DBMS responds, the tool parses the result and delivers to the user. The result can be returned in CSV (Comma Separated Values) or JSON (Javascript Object Notation) formats.

The format below depicts a generic request used to produce a *RFQ* query.

```
http://tool.domain/v1/data?
   metrics=METRIC_1,METRIC_N
   &dimensions=DIMENSION_1,DIMENSION_N
   &filters=CLAUSE_1_FILTER_1,CLAUSE_1_FILTER_N;
   CLAUSE_N_FILTER_1,CLAUSE_N_FILTER_N
   &format=json
```

First, it has the domain and main route *tool.domain/v1/data* to the API call. Then, it is formed by four parameters:

- **metrics:** the list of metrics separated by commas. At least one metric need to be specified;
- **dimensions:** the list of dimensions separated by commas; it supports zero or more dimensions;
- **filters:** the list of clauses separated by semi-colon (*AND* operation); in each clause, a list of filters are separated by commas (*or* operation). it supports zero or more combination of filters. Each filter supports comparison operations for numeric values ("==" , "≥", "≤", "<", ">" or "!=") and for string values ("==", "!=");
- **format:** it has two options: *csv* or *json* (default), to specify the output format.

It has a one-to-one counterpart in a *RFQ* request, making the translation straightforward. In other words, there are a set of metrics, dimensions, and clauses in CNF.

In order to integrate the tool with existing databases, it is required to create a database schema description, which is a set of files listing the available relations, metrics and dimensions. It uses its own schema description format, which is a YAML document describing the *mapping* of the *RFQ* elements into the database elements. An example of YAML file is shown below. It contains one aliases, which correspond to an existing concrete view. It contains as well one metric description and the list of dimensions (if any). They have same name of the target column names.

```
alias: "student"
  dimensions:
    - "region"
    - "grade"
    - "city"
metrics:
    - "st_id"
```

This description necessary because the *SQL* schema does not contain all the meta-data required to use *RFQ*. Such description is used by the query rewriting algorithm, to convert the query in the virtual schema to the real database. An additional parameter (`format`) enables to choose how the result is shipped.

As the tool is not a new DBMS, but an abstraction layer, it was not required implement a new DBMS to use *RFQ*, it can be used with existent DBMS. At the moment of the writing of this paper, the tool can be used with two DBMS, MonetDB and PostgreSQL.

4 CASE STUDY

The case study consists of tabular Open Data (CSV files) extracted from public data sources and integrated into a single database instance. The public data sources contain information about the educational system of Brazil, covering all educational levels. It also contains data about public policies for providing internet access (digital inclusion) for the citizens. It contains 24 tables, 1169 attributes and approximately 2.5 billion records [3].

The integrated data source is called *Blended Integrated Open Data (BIOD)*[4]. The microservice request is forwarded to the *BlenDb* middleware, which translates the URL into SQL and calls the integrated database API. The middleware is implemented using *Node.js*.

We integrate these data sources under the *MonetDb*[5] column store, using traditional Extract Transform and Load (ETL) approaches, by executing data extraction scripts. This is a prior step, which details are out of the scope of the *RFQ* representation.

[3]The complete database schema can be found at: https://gitlab.c3sl.ufpr.br/simmctic/biod/biod-database.

[4]The BIOD microservice is available at https://biod.c3sl.ufpr.br/index_en.html

[5]https://www.monetdb.org/

We define the metrics and dimensions, create the virtual schema and the configurations files. These specifications are needed to be able to translated the API calls into SQL. It resulted in a database with 682 metrics and 904 dimensions which can be queried. The original degree of normalization of the data sources was maintained, to avoid the creation of additional maintenance scripts. This means we do not create a new dimensional model for each integrated source. We now present two query examples to illustrate the applicability of the approach. It involves analytic queries with different targets.

Query 1. *Let us suppose we want to retrieve general descriptions about the city of Curitiba, with information about population, economic indicators, internet connection, among others. The RFQ query has the following definition:*

Q *(SumPopulation, CountSchools, CountUniversity, AvgEconomyGDP, AvgEconomyIncomeLevel, CountSimmcPoint) (State, Region) (CityName = Curitiba, Year = 2017)*

The query can be done through the following API call [6]

```
http://tool.domain/v1/data?
   metrics=sumpopulation,countschools,
   countuniversity,sumeconomigdp,
   avgeconomyincomelevel,countsimmcpoint
   &dimensions=region,state
   &filters=cityname==Curitiba;year==2017
```

The query response has data about the city: the population, the number of high schools and universities, the value of Gross Domestic Product, the income level, the number of active internet connection, and the given region and state.

The query above could be done using SQL. The user would need to develop a query joining and projecting all the attributes over six relations (Population, Schools, Universities, GDP, City and Point) and to restrict the year and name of the city in each one. The SQL query is:

```
SELECT
  SUM(pop.POPULATION), COUNT(sc.SC_ID) AS N_SCHOOL,
  COUNT(un.UN_ID) AS N_UNIVERSITY, SUM(gdp.GDP),
  AVG(gdp.INCOME_LEVEL), COUNT(poi.POI_ID) AS N_POINT,
  c.REGION, c.STATE
FROM POPULATION pop
  INNER JOIN SCHOOL sc ON sc.CITY_ID = pop.CITY_ID
  INNER JOIN UNIVERSITY un ON un.CITY_ID=pop.CITY_ID
  INNER JOIN GDP gdp ON gdp.CITY_ID = pop.CITY_ID
  INNER JOIN POINT poi ON poi.CITY_ID = pop.CITY_ID
  INNER JOIN CITY c ON c.CITY_ID = pop.CITY_ID
WHERE c.NAME = "Curitiba" AND
  sc.YEAR = 2017 AND un.YEAR = 2017 AND
  gdp.YEAR = 2017 AND inc.YEAR = 2017 AND
  poi.YEAR = 2017 AND pop.YEAR = 2017
GROUP BY c.REGION, c.STATE
```

[6]We have translated the name of the elements to ease the comprehension.

Query 2. *We consider a specific scenario about schools, group by region of Brazil and administrative dependency of school (federal, state, municipal, private). The query produced is:*

Q *(CountSchools, AvgSchoolClassroom, AvgSchoolEmployees) (Region, SchoolAdministrativeDependency) (Year = 2017)*

The answer contains the number of schools, average of used classrooms and average of employees, grouped by region and administrative dependency. The equivalent SQL query is:

```
SELECT COUNT(sc.SC_ID) AS N_SCHOOL,
  AVG(sc.SC_CLASSROOM), AVG(sc.SC_EMPLOYEES),
  sc.SC_ADMIN_EPENDENCY, c.REGION
FROM SCHOOL sc
  INNER JOIN CITY c ON c.CITY_ID = sc.CITY_ID
WHERE sc.YEAR = 2017
GROUP BY c.REGION, sc.SC_ADMIN_DEPENDENCY
```

The corresponding API call is the following:

```
http://tool.domain/v1/data?
  metrics=countschools,avgschoolclassroom,
  avgschoolemployees&dimensions=region,
  schooladmdependency&filters=year==2017
```

The design of analytic queries often involve several data sets and typically rely on user knowledge about each relations containing the data. The utilization of relation-free API calls frees the user/developer from having to locate the tables; instead, the users only choose over a set of valid dimensions and metrics. It also frees the users from understanding the details of each table in order to manually join them. In addition, it could be exported to web applications.

Our approach simplifies the development of queries involving several relations and attributes regarding to *SPJG* queries. This can be explained because the users choose over a set of possible dimensions and metrics, and the presented solution translates the user interest into a *SPJG* query. Thus our key contribution is an approach that provides easy access over tabular *Open Data* sources, through a simple query representation, translating automatically any valid element combination into *SQL*.

The integration of the data sources is done through a set of extraction tasks and by the specification of the mapping files. While this is a time consuming and laborious step, it follows existing data integration approaches. The mapping process of attributes and dimensions into the schema description was done manually, where a full understanding of the data model was necessary. The JOINs do not need to be specified, but the name of *joinable* attributes need to be the same. We consider that an automated approach, for instance adapting schema matching solutions, would enable potential benefits, thus is an interesting point to future research.

There are additional aspects to be considered. Without the real database schema, is not possible to calculate efficiency measures such as the number of joins performed or relations used. These measures are dependent of how the real database is structured. For instance, consider two databases with the same data, but one in the third normal form and one in the first. Both would have the same set of metrics and dimensions, while the same *RFQ* query in both would return the same data, the query structure would differ, also the number of joins and relations used as a third normal form demands more relations and joins. In our case study, we handle poorly normalized databases.

Our approach relies in the process of creating a simplified virtual schema, which allows the elaboration of simpler queries. To use the simplified virtual schema, *RFQ* is not required. A user could wonder if there is any advantage in *RFQ* if the virtual schema already simplifies the query elaboration. The main advantage of *RFQ* is the access to data through an API without naming the relations and JOINs, which is not case if developing the queries directly in SQL.

The goal is to provide easy analytic querying capabilities, thus the representation does not cover any kind of format involving *SPJG* queries. This means it would not be possible to express, for instance, all TPC-H or TPC-B queries, or other kinds of JOINs. This is a chosen limitation of the approach. In cases where more expressive *SQL* would be needed, existing BI solutions could be used instead.

By narrowing the kinds of queries, the query generation task becomes simpler, producing always the same style of output queries, *SPJG* with INNER JOINs. While the number of illustrated *Joins* is not large in this specific case, it can be generalized for larger queries, since the number of metrics available is important. The framework is a middle-layer between the API and the target database, not a specific database component, thus any query optimization need to be performed by the target DBMS. The approach is currently being used in a real world scenario, showing good empirical applicability, but it is not yet possible to provide a quantitative assessment for such a modeling task. Detailed studies about user acceptation need to be conducted. Its larger success depends on future adoption in other solutions.

5 RELATED WORK

The large availability of *Open data* sources has raised opportunities in several research subjects, such as: data integration and exchange, data analytics visualization and languages, data analytics *RESTful API*, and others. Our solution can be placed as an analytic query representation format, which is accessible through a RESTful API.

Focusing on analytics applications, *Kwakye* (Kwakye et al., 2013) presents the integration of disparate data marts, concentrating on how to find the correspondences and to merge the data sources. The concrete queries are done using standard *SQL*. The *Cheetah* framework (Chen, 2010) presents the idea of creating one virtual view per fact table. Once the views are defined, it is necessary to find the correct relations and to join them using SQL.

There are different approaches providing keyword-based query languages, such as the SODA system (Blunschi et al., 2012), SQAK (Tata and Lohman, 2008) or the work from (Bergamaschi et al., 2011). They provide ways to ease the task of querying over several sources. These approaches have as main goal to build the queries from the keywords, often in a exploratory way, using additional structures as support. For instance, SODA and SQAK use metadata to iteratively explore a datawarehouse and to create queries. This enables to ease the query construction, but due to its interactive nature, it is more adapted to online data finding than to making API calls.

The Schema-free SQL approach (Li et al., 2014) enables writing partial (or complete) SQL to answer questions. It does not require to specify relations, and the attributes specification may be incomplete. The approach return a top-K set of queries to be executed.

The most common approach is to use Business Intelligence frameworks, such as *Saiku*, *IBM Cognos*, *QLikView*, or many others. They provide complete frameworks accessible for the developers or data analysts, often providing graphical interfaces or specialized languages. These frameworks enable constructing complex analytical queries for any domain. When using the graphical interface, it is important to have the dimensional model clearly defined, so the features of the tools can be fully explored. Many of them translate the dimensional specifications into the MDX (Multidimensional Expressions) language, which enables writing analytical queries. The final user often does not need to write queries in MDX. Each framework provide its own interface prior to the query translation task.

Other approaches, considering having an API as upper layer, propose to generate queries from *RESTful APIs*. In (Ed-douibi et al., 2018) an UML class diagram is used to generate a database schema and a *RESTful API*, respecting the OData pattern. It is designed to OLTP queries, and there are extensions to use OLAP queries. The translations are direct query expansions. In (Sellami et al., 2014) an *RESTful API* is presented as a communication language with relational and NoSQL data stores. In this case, the join operations and aggregation functions are not allowed, which means they are restricted, enabling to design simple queries, but not targeted to analytics.

6 CONCLUSIONS

We presented a solution to ease the task of creating analytic queries on integrated tabular *Open Data* sources. We describe our query representation format, called *Relation-Free Query* (*RFQ*), where we do not explicitly define the relations and the joins, enabling to focus on the attributes, metrics and dimensions. The RFQ requests are done through a *RESTful API*. We provide a virtual global schema with the information about possible dimensions, metrics and attributes, which is mapped into the target tables of the concrete database.

We presented a case study, that consists on querying over integrated Open Data sources, having more than 900 dimensions and 600 metrics. The queries are created using the developed *RESTful API*, which has a one-to-one correspondence with the *RFQ* format (*dimensions, metrics, filters*). This enables to have a published Open Data querying microservice. A microservice-based architecture was chosen because it enables easy access by mobile or web application developers, thus aiming at providing a public accessible service. A current version of the service, is freely available online in a real world scenario, called BIOD (Blended Integrated Open Data).

As future work, we plan to provide adapters of the queries generated by the tool to enable querying on different data lakes and to develop a solution to find *unionable tables* to integrate the sources.

ACKNOWLEDGMENTS

We would like to thank the Simmc/UFPR and SNPPIR/UFPR projects, which partially funded this work.

REFERENCES

Abadi, D. and Stonebraker, M. (2015). C-store: Looking back and looking forward. Talk at VLDB - Very Large Database Systems.

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc.

Aligon, J., Golfarelli, M., Marcel, P., Rizzi, S., and Turricchia, E. (2014). Similarity measures for OLAP sessions. *Knowl. Inf. Syst.*, 39(2):463–489.

Bergamaschi, S., Domnori, E., Guerra, F., Trillo Lado, R., and Velegrakis, Y. (2011). Keyword search over relational databases: A metadata approach. In *Proceedings of the 2011 ACM SIGMOD*, SIGMOD '11, pages 565–576, New York, NY, USA. ACM.

Blunschi, L., Jossen, C., Kossmann, D., Mori, M., and Stockinger, K. (2012). Soda: Generating sql for business users. *Proceedings of the VLDB Endowment*, 5(10):932–943.

Chen, S. (2010). Cheetah: a high performance, custom data warehouse on top of mapreduce. *Proceedings ofVLDB*, 3(1-2):1459–1468.

Ed-douibi, H., Izquierdo, J. L. C., and Cabot, J. (2018). Model-driven development of OData services: An application to relational databases.

Kwakye, M. M., Kiringa, I., and Viktor, H. L. (2013). Merging multidimensional data models: a practical approach for schema and data instances. In *Proceedings of the 5th DBKDA*, pages 100–107.

Lenzerini, M. (2002). Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246.

Li, F., Pan, T., and Jagadish, H. V. (2014). Schema-free sql. In *Proceedings of the 2014 ACM SIGMOD*, SIGMOD '14, page 1051–1062, New York, NY, USA. ACM.

Miller, R. J. (2018). Open data integration. *Proc. VLDB Endow.*, 11(12):2130–2139.

Richardson, L., Amundsen, M., and Ruby, S. (2013). *RESTful Web APIs: Services for a Changing World.* " O'Reilly Media, Inc.".

Sellami, R., Bhiri, S., and Defude, B. (2014). Odbapi: a unified rest api for relational and nosql data stores. In *2014 IEEE International Congress on BigData*, pages 653–660. IEEE.

Tata, S. and Lohman, G. M. (2008). Sqak: Doing more with keywords. In *Proceedings of the 2008 ACM SIGMOD*, SIGMOD, pages 889–902, New York, NY, USA. ACM.