Facilitating the exploitation of Linked Open Statistical Data: JSON-QB API requirements and design criteria

xxx ddd and yyy sss

No Institute Given

Abstract. Recently, many organizations have opened up their data through open data portals. Some of the portals adopt Linked Data, a promising paradigm for opening data that facilitates data integration on the web. A major part of these data concern statistics and standard vocabularies have been proposed to enable their modelling as Linked Open Statistical Data (LOSD). Although LOSD potential is high, their exploitation is low for two reasons. First, the development of LOSD applications requires skills (RDF, SPARQL) not popular at web developers and second many portals adopt different publishing practices, thus hampering their interoperability. Thus, in order to unleash the full potential of LOSD there is a need to standardize the interaction (i.e. input, output and functionality) and hide most of the complexity. Towards this end, this paper describes the requirements and design criteria of a JSON-QB API that i) facilitates the development of LOSD tools through a style of interaction familiar to web developers and ii) offers a uniform way to access LOSD by hiding data discrepancies. A proof of concept implementation of the JSON-QB API demonstrates the proposed functionality.

1 Introduction

Recently, many governments, organisations and companies are opening up their data for others to reuse through *Open Data* portals [11]. These data can be exploited to create added value services, which can increase transparency, contribute to economic growth and provide social value to citizens [8].

A major part of open data concerns statistics (e.g. economical and social indicators) [2]. These data are often organised in a multidimensional way, where a measured fact is described based on a number of dimensions. In this case, statistical data are compared to data cubes. Thus, we onwards refer to statistical multidimensional data as *data cubes* or just *cubes*.

Linked data has been introduced as a promising paradigm for opening up data because it facilitates data integration on the Web [1]. Concerning statistical data, standard vocabularies such as the RDF data cube (QB) vocabulary[4], SKOS[14] and XKOS[3] enable modelling data cubes as Linked Open Statistical Data (LOSD).

Although LOSD potential is high, their exploitation is currently low for two reasons. First, using LOSD requires skills and tooling (e.g. RDF, SPARQL) that are less widespread than some other web technologies (e.g. JSON, Javascript). For example, there are many Javascript visualization libraries that consume JSON data (e.g D3.js, charts.js), while there are just a few that consume RDF and their functionality is limited. Second, many portals that use the standard vocabularies (QB, SKOS, XKOS) often adopt different publishing practices [9], thus hampering their interoperability. As a result it is not easy to create generic software tools that can be reused across LOSD. Usually, developed tools assume that data are published only in a specific form.

In order to unleash the full potential of LOSD there is a need to standardize the interaction (i.e. input, output and functionality) with LOSD in a way that facilitates the development of reusable software. Towards this end, this paper describes the requirements and design criteria of a JSON-QB API that aims to exploit the advantages of linked data (e.g. easy data integration) while making data available in a structure and format that is familiar to a larger group of developers. Some of the flexibility, and associated complexity, of linked data is removed, in favour of simplicity and ease of use. Moreover, the API offers a uniform way to access the data by hiding any data discrepancies, thus enabling the development of generic software tools that can be reused across datasets.

The rest of the paper is organized as follows, section 2 presents the motivation for the development of a JSON-QB API, section 3 describes the methodology followed to define the requirements and design criteria presented at section 4. Section 5 presents a proof of concept implementation of the API. Finally, section 6 draws conclusions.

2 Motivation

Currently, many LOSD have been made available on the Web through official portals launched by private or public bodies that own the data. For example, the European Commission's Digital Agenda provides its Scoreboard[1] as LOSD. Census data of 2011 from Ireland[2] and Italy[3] have been published as linked data by their National Statistics Institutes. The Department for Communities and Local Government (DCLG)[4] in the UK, the Scottish Government[5], and the Statistics Bureau of Japan[6] opened up their statistics as linked data.

Although the above portals use the same standard vocabularies, they often adopt different publishing practices. As a result, generic tools that operates across LOSD datasets cannot be created. However, tools for exploiting LOSD have already been developed which assume data published only in a specific way. For example, existing tools enable: i) the browsing of LOSD e.g. Data Cube faceted browser[6], CODE Query wizard[7] ii) the performance of OLAP

[1] http://digital-agenda-data.eu/data
[2] http://data.cso.ie
[3] http://datiopen.istat.it
[4] http://opendatacommunities.org/data
[5] http://statistics.gov.scot
[6] http://data.e-stat.go.jp

operations like roll-up/drill-down, slice, dice e.g. OpenCube OLAP Browser [12], QB2OLAP[16] iii) the performance of statistical analysis on LOSD e.g. OpenCube R statistical analysis tool[10] and iv) the visualization of LOSD e.g. CubeViz[13], StatSpace[5].

Except from the exploitation tools, complete platforms (e.g. PublishMyData[7]) exist that aim both in to publishing and exploiting LOSD. In this case, published data can be consumed only by tools of the same platform, since different publishing practices are adopted. This lead to the creation of LOSD system silos (software & data) that cannot interoperate among each other.

All the above tools and platforms follow the same traditional architecture (figure 1) where each tool has an integrated access layer. If several tools are created for the same portal (i.e. same publishing practices), then each tool has to develop separately a similar data access layer. In addition, if a tool has to be used at another portal, then a new data access layer has to be created leading to additional costs. More importantly, the development of data access layers requires significant programming expertise in LOSD, a skill that is not widely available between developers.

As a result, there is a need to standardize the interaction with LOSD in a way that hides the LOSD complexity to the developers and offers a uniform way to access the data hiding any data discrepancies.

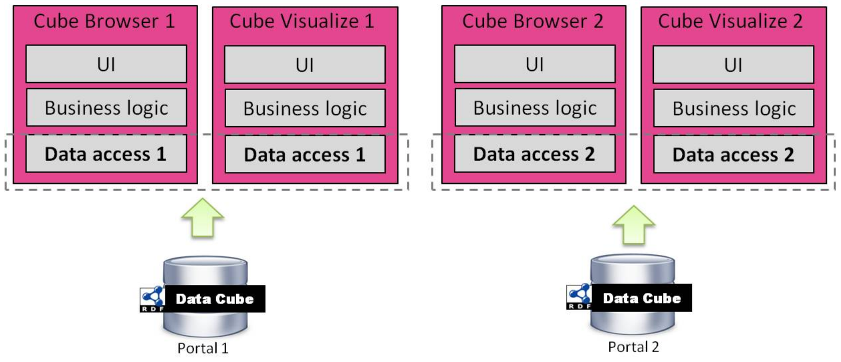

Fig. 1. Traditional architecture for LOSD tools

3 Methodology

In order to achieve the objectives of the paper we adopt the following methodology:

[7] http://www.swirrl.com/

– Study the related work. We focus on: i) APIs that facilitate the interaction with data cubes and ii) data formats that can be exploited to represent the result (i.e. output) of the API.
– Collect user requirements from developers that currently create applications for LOSD.

Currently, there exist many APIs that facilitate the interaction with data cubes. These APIs offer basic functionality that covers the cube's logical model, but they also support more advanced OLAP operations including aggregations, slicing, roll-up/drill-down etc. For example, the Oracle OLAP Java API [15] allows users to select, explore, aggregate, calculate, and perform other analytical tasks on data stored in an Oracle data warehouse. Olap4j[8] is another Java API for accessing data cubes, which is compatible with many OLAP servers (e.g. Mondrian, Palo and SAP Business Warehouse). It enables the browsing of meta-data including the cubes, dimensions and hierarchies. Olap4j also supports Multidimensional Expressions (MDX) that is the query language for OLAP.

There are also some REST APIs with similar functionality. The Data Brewery[9] offers a set of Python tools, including a REST API, for processing and analysing data cubes stored at a relational data base (e.g. MySQL, PostgreSQL). Apache Lens[10] is an analytics platform that integrates Hadoop with traditional data warehouses (e.g. Apache Hive, Amazon Redshift). Lens provides a REST API to handle data cubes, that also supports "OLAP Cube QL" - a high level SQL like language to query data organized as data cubes.

All the above APIs handle data cubes that are stored in traditional databases or data-warehouse. However, none of them can handle data cubes stored as linked data using the QB vocabulary.

Regarding the output of the REST APIs, JSON is a commonly used simple format. Existing REST APIs use case-specific JSON responses, however JSON extension formats have already been proposed to model data cubes and linked data. Specifically, JSON-LD[11] offers a method for encoding linked data using JSON. While, JSON-stat[12] is a JSON format for modelling linked statistical data, however the structure is too complicated when it comes to simple visualizations (e.g. maps). Moreover, it has some limitations e.g. does not support pagination of results.

Finally, to collect the user requirements, we established a continuous discussion with developers that currently create applications for LOSD. The discussion mainly occurs within the EU funded project OpenGovIntelligence[13], that aims to exploit LOSD for improving the public services. To facilitate the collection of requirements we organized a dedicated workshop in Manchester with participation of relevant developers.

[8] http://www.olap4j.org
[9] http://databrewery.org/
[10] lens.apache.org
[11] https://json-ld.org
[12] https://json-stat.org/
[13] http://www.opengovintelligence.eu/

4 Requirements and design criteria

This section presents the requirements and design criteria related to the JSON-QB API. Generally, the API should follow patterns and practices familiar to "mainstream" web developers, which facilitate the creation of visualisations and interactive applications that use the data. Moreover, it should be suitable for use by a wide range of statistics publishing organisations, so that data users can have a standard interface to LOSD. This will put constraints on the way that publishers manage their data, however those constraints should be reasonable and manageable.

4.1 Search data cubes

The LOSD cloud currently contains many data cubes and their number still increases. Thus, applications need to search for cubes based on some criteria. For example, get cubes that measure unemployment, or get cubes for Greece. The search can be even more complex e.g. get cubes about unemployment in Greece after 2010. To fully exploit the searching functionality, the JSON-QB API ideally should search over the LOSD cloud and not be limited to a single RDF store.

The search functionality can also be extended to support not only user specific criteria (as the previous examples), but also support the "automatic" search of compatible cubes that could be processed together. For example, having a cube at hand search for other cubes that are compatible for combined statistical analysis, for visualisation or for browsing. The cube compatibility criteria is still an open issue and is out of the scope of this paper.

4.2 Explore data cube structure

Once a cube has been identified (e.g. through the search functionality) the processing application (e.g. cube browser) needs to initialize the user interface or the analysis with information related to the cube structure. For example, populate drop-down menus with the cube dimensions and measures. The QB vocabulary clearly identifies the main elements of the structure that should be accessed through the JSON-QB API:

- Dataset meta-data. They include information like the label, description, issue date, publisher and license.
- Dimensions. They include all the dimension properties of the cube (e.g. reference area, reference period).
- Measures. They include all the measure properties of the cube (e.g. unemployment, poverty)
- Attributes. They include all the attribute properties of the cube (e.g. unit of measure)
- Dimension values. They include all the values of a dimension (e.g. male, female) that appear at the cube.

– Dimension levels. In the case of hierarchical data, dimension values are organized to hierarchical levels (e.g. region, district).
– Attribute values. They include all the values of an attribute (e.g. euro, dollar) that appear at the cube.

Regarding the last three elements, the QB vocabulary does not offer a way to retrieve the values / levels directly from the structure. Thus the API should iterate over the cube observations, which is a time consuming task.

4.3 Slicing and filtering

There are already methods available for downloading entire data cubes but people often want just small parts. Whole cubes are often too big to be well-suited to interactive applications, and if the data updates frequently, then it's important for people to be able to retrieve up-to-date extracts of the data, rather than keeping their own copies of full datasets up to date. The JSON-QB API should allow applications to take exactly the data they want by defining constrains (i.e. filters) to the dimension values. It should support many filtering options including:

– Single values e.g. refPeriod=2010.
– Multiple values e.g. refPeriod=[2010, 2011, 2012]
– Ranges e.g refPeriod=[2010 ... 2015]
– Greater/smaller than e.g. refPeriod>2010
– Hierarchical data filtering e.g. refArea="all council areas in Scotland"

In many cases, applications do not need all the requested data at once, because they process them at bunches. For example, a cube browser shows a part of the data allowing the user to navigate to the previous/next page of data. Thus, the JSON-QB API should support paging and ordering of the results. The ordering of the results can be in ascending or descending order based on a dimension. However, in some cases this is a complicated task e.g. ordering based on 2 ore more dimension. Moreover, lexicographical ordering is not always appropriate (e.g. for the days of the week), thus other types of ordering should be applied.

4.4 Ease of use

Linked data offer many benefits to web developers, including the easy integration on the web. However, linked data technologies (i.e. RDF, SPARQL) are unfamiliar to many developers, thus creating many obstacles at their adoption. The purpose of the JSON-QB API is to exploit the advantages of linked data through a style of interaction that is familiar to web developers, thus helping them create data visualisations and applications. It is not necessary for the API to be a complete "round-trippable" representation of the data, it is acceptable to lose some information in favour of greater ease of use.

The ease of use of an API is related both to the input of the API (API calls) and the output. Regarding the input of the API there are mainly two design options: i) use a separate parameter for each required input and ii) model all the required input as a JSON object. The first option is most popular at existing APIs, while the second is more flexible and expressive. For example, it enables the expression of relations other than equality e.g. greater than (see Table 1).

Table 1. Input of API: separate parameters vs JSON object

Separate parameters	**Input as JSON**

`GET /slice?dataset=home-care&`
`gender=male`
	`{"dataset": "home-care",`
	` "filter": {`
	`   "gender": "male"`
	`}}`

`Cannot be expressed`	`{"dataset": "home-care",`
	` "filter": {`
	`   "gender": "male"`
	`   "age": { "greater-than": 50}`
	`}}`

Regarding the output of the API, JSON is a popular, easy to use format. Usually, applications and visualizations do not require an n-array/ tabular response (e.g. JSON-stat); an array of observations is sufficient and more straightforward. In case that a tabular response is required, then it can easily be constructed from the observations. While JSON-QB API aims in hiding some of the complexity of linked data, responses should include URIs as identifiers of key entities, to retain the connection to data on the web and to support reliable combining of data from different sources within a data consuming application.

4.5 Uniform data access

Currently many LOSD have been published, however a lot of them adopt different publishing practices. The JSON-QB API should work on top of any of these data, offering uniform access to the data and hide any discrepancies. Obviously, this will require separate implementations to comply with the different publishing practices.

Ideally, the standardization of the JSON-QB API specification will also contribute to the formulation of an application profile for the QB vocabulary. The profile will include best practices that can be used by data publishers to provide data in a compatible way, facilitating in this way the development of generic LOSD tools. This will add some constraints on the way that publishers manage their data, however those constraints will lead to greater exploitation of the data.

4.6 High performance

The volume of LOSD is big, reaching the magnitude of million triples per cube. Thus, SPARQL queries that iterate over all the observations tend to be slow. For example, a query to get all the dimension values that appear at a cube needs to iterate over all the observations. As a result, applications that use such queries seem to be non-responsive.

The JSON-QB API can improve performance of demanding SPARQL queries through efficient caching of the responses. The caching policy (e.g. Least Recently Used, Least Frequently Used) plays an important role at the performance improvement. Note that caching of API responses is much easier that caching of arbitrary SPARQL queries. Allowing a SPARQL query to run on a collection of data means that if any of the data changes, it is possible that the query response changes. It is complex to analyse which queries touch a particular data cube or particular part of the data, thus making cache clearing difficult. With the API call, most requests will return data from individual data cubes, so it is easier to know which cached responses must be invalidated when data is updated.

Another task that can improve the performance of the API is the pre-computation of aggregations: i) across a dimension of the cube e.g. compute the SUM of the sales over time and thus ignore the time dimension of the cube and ii) across a hierarchy e.g. if a cube contains the election results at municipality level, then aggregations can be computed at region and at country level. The pre-computation of the aggregations facilitates the execution of queries, because there in not the need to compute the aggregations on-the-fly when requested.

4.7 Extensibility

Finally, the JSON-QB API should be extensible, thus take future growth into consideration minimizing the effort required for the extension. Extensions can be implemented through the: i) addition of new functionality e.g. while the initial aim is to build an API on top of RDF databases, other kinds of database could be used, enabling flexibility and innovation and ii) modification of existing functionality e.g. support modified search (see section 4.1) of filtering options (see section 4.3).

5 Proof of concept implementation

The architecture of the JSON-QB API (figure 2) is simple and is developed as a middle-ware between LOSD and the applications that consume the data. The API receives the REST calls and translates them to SPARQL queries which are executed at LOSD portals. Then, the returned results are transformed to JSON format that can easily be consumed by applications. Table 2 presents example API calls, the corresponding SPARQL queries and the returned JSON results.

We have developed a proof of concept implementation of the JSON-QB API[14] which can be installed on top of existing RDF repositories that store data using the QB vocabulary. It uses existing technologies including the Jersey framework[15] for the implementation of the RESTful services, the Rdf4j[16] Java frame-

[14] https://github.com/OpenGovIntelligence/json-qb-api-implementation
[15] https://jersey.github.io/
[16] http://rdf4j.org/

Table 2. Example API calls

Get cube Dimensions	
API call	`GET /dimensions?dataset=http://example.com/cube/unemployment`
SPARQL query	`PREFIX qb: http://purl.org/linked-data/cube#`
	`PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#`
	`PREFIX ex: http://example.com/cube/`
	`select distinct ?dim ?label where {`
	`    ex:unemployment qb:structure ?dsd.`
	`    ?dsd qb:component ?comp.`
	`    ?comp qb:dimension ?dim.`
	`    ?dim rdfs:label ?label.}`
JSON result	`{"dimensions":[`
	`  {"@id":  "http://example.com/dimension/refArea",`
	`   "label":"Reference area"},`
	`  {"@id":  "http://example.com/dimension/refPeriod",`
	`   "label":"Reference period"},`
	`  {"@id":  "http://example.com/dimension/ageGroup",`
	`   "label":"Age group"},`
	` ]`
	`}`
Get a cube Slice	
API call	`GET /slice?dataset=http://example.com/cube/unemployment&`
	`           http://example.com/dimension/refPeriod=2016`
SPARQL query	`PREFIX qb: http://purl.org/linked-data/cube#`
	`PREFIX ex: http://example.com/cube/`
	`select ?obs ?refArea ?ageGroup ?unemployment where {`
	`       ?obs qb:dataSet ex:unemployment.`
	`       ?obs ex:refPeriod "2016".`
	`       ?obs ex:refArea ?refArea.`
	`       ?obs ex:ageGroup ?ageGroup.`
	`       ?obs ex:unemploymentRate ?unemployment.`
	`}`
JSON result	`{"observations":[`
	`      {"@id":"http://example.com/observation/1",`
	`        "refArea"         : "Greece",`
	`        "ageGroup"        : "25-34",`
	`        "unemploymentRate": 0.34`
	`       },`
	`       {"@id":"http://example.com/observation/2",`
	`        "refArea"         : "Greece",`
	`        "ageGroup"        : "35-44",`
	`        "unemploymentRate": 0.29`
	`       },.... ]`
	`}`

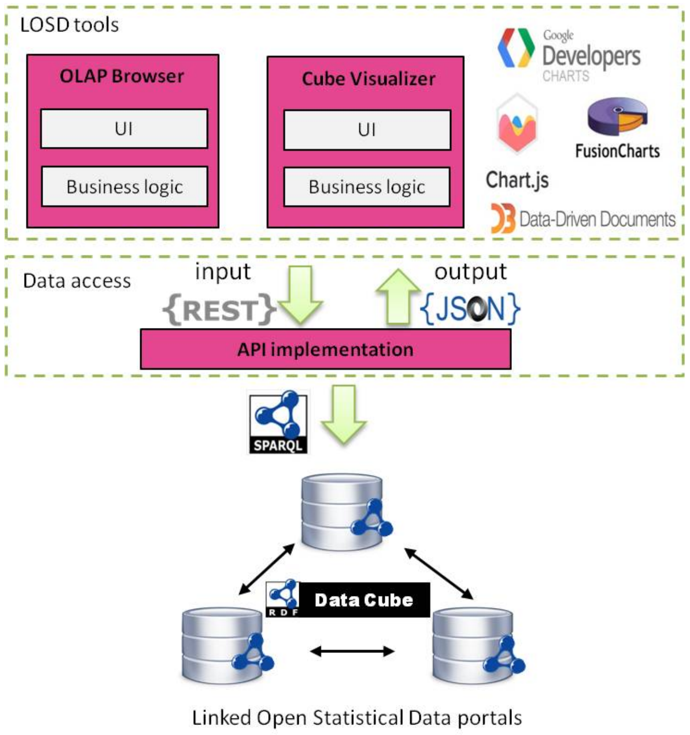

Fig. 2. JSON-QB API architecture overview

work for processing RDF data and the Gson[17] Java library to serialize Java Objects into their JSON representation. The API is still under development and the current version implements only a subset of the proposed functionality.

In order to demonstrate the use of the JSON-QB API, we developed the "Cube Visualizer" (figure 3), an application that consume data produced by the JSON-QB API to presents a graphical representations of LOSD. Users choices (e.g. measure and dimension to visualise) are translated to appropriate API calls. The returned JSON results are presented to the user in the form of a chart e.g. bar chart, pie chart, area chart. The development of this application was easy, because there is no need to implement a data access layer, the required data are acquired through the API.

6 Conclusion

Currently, the LOSD cloud contains many datasets and their number increases. However, their exploitation still remains low due to two reasons. First, skills and

[17] https://github.com/google/gson

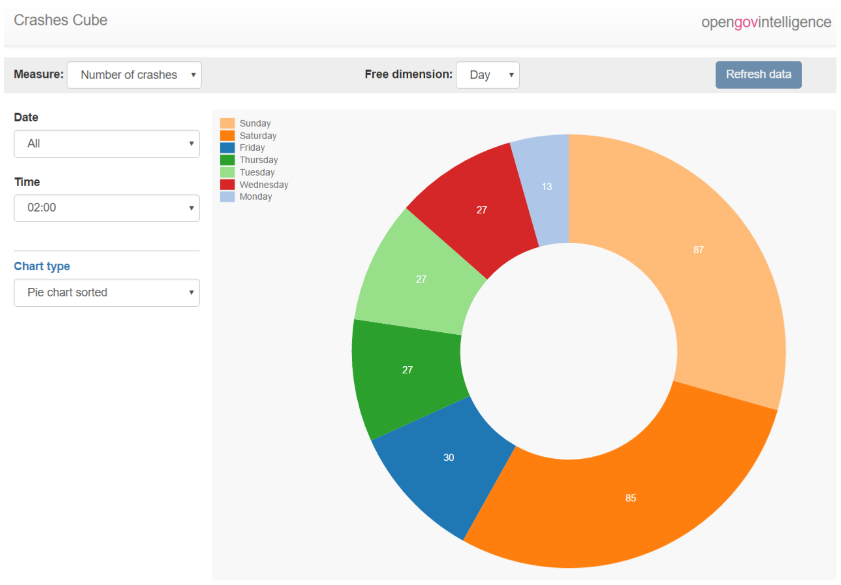

Fig. 3. Cube Visualizer

tooling for linked data are not widespread among developers and second, existing portals adopt different publishing approaches, thus hindering the development of tools that can operate across LOSD datasets.

In this paper we describe the requirements and design criteria of a JSON-QB API that standardises the interaction with LOSD aiming at their broader exploitation. Specifically, the API facilitates the development of LOSD tools through a style of interaction familiar to web developers. It also provides uniform access to LOSD data hiding any data discrepancies. However, in order to achieve the uniform data access, either different implementation of the API should be created (one for each set of publishing practices) or a set of best practices should be adopted by publishers to provide data in a compatible way. We anticipate that the standardization of the JSON-QB API specification will contribute towards the formulation of these best practices.

Future work includes the further development of the API based on the specified requirements and design criteria. Moreover, there are open issues related to the cube compatibility criteria, the ordering of the API results and the expression of the API's input either as separate parameters or as a JSON object. Finally, we should note that while our objective is to provide an API on top of a triple store holding linked data, the same API could be implemented with other approaches to storing data.

Acknowledgments. Part of this work was funded by the European Commission within the H2020 Programme in the context of the OpenGovIntelligence project (http://OpenGovIntelligence.eu) under grant agreement no. 693849.

References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
2. Capadisli, S., Auer, S., Ngonga Ngomo, A.C.: Linked sdmx data. Semantic Web 6(2), 105–112 (2015)
3. Cotton, F.: XKOS an skos extension for representing statistical classifications (unofficial draft). Tech. rep., DDI Alliance (January 2017)
4. Cyganiak, R., Reynolds, D.: The RDF data cube vocabulary: W3C recommendation. Tech. rep., W3C (January 2014)
5. Do, B.L., Wetz, P., Kiesling, E., Aryan, P.R., Trinh, T.D., Tjoa, A.M.: Statspace: A unified platform for statistical data exploration. In: OTM Confederated International Conferences, Rhodes, Greece, October 24-28. pp. 792–809. Springer International Publishing (2016)
6. F, M., G, S., N, L.: A dynamic faceted browser for data cube statistical data. In: W3C workshop on Using Open Data (2012)
7. Hoefler, P., Granitzer, M., Veas, E.E., Seifert, C.: Linked data query wizard: A novel interface for accessing sparql endpoints. In: Workshop on Linked Data on the Web (LDOW) (2014)
8. Janssen, M., Charalabidis, Y., Zuiderwijk, A.: Benefits, adoption barriers and myths of open data and open government. Information Systems Management 29(4), 258–268 (2012), http://dx.doi.org/10.1080/10580530.2012.716740
9. Kalampokis, E., Roberts, B., Karamanou, A., Tambouris, E., Tarabanis, K.: Challenges on developing tools for exploiting linked open data cubes. In: Proceedings of the 3rd International Workshop on Semantic Statistics (SemStats2015) within the 14th International Semantic Web Conference (ISWC2015). vol. 1551. CEUR-WS (2015)
10. Kalampokis, E., Nikolov, A., Haase, P., Cyganiak, R., Stasiewicz, A., Karamanou, A., Zotou, M., Zeginis, D., Tambouris, E., Tarabanis, K.: Exploiting linked data cubes with OpenCube toolkit. In: Proc. of the ISWC 2014 Posters and Demos Track a track within 13th International Semantic Web Conference (ISWC2014), vol. 1272. CEUR-WS (2014)
11. Kalampokis, E., Tambouris, E., Tarabanis, K.: A classification scheme for open government data: towards linking decentralised data. Int. J. Web Eng. Technol. 6(3), 266–285 (Jun 2011), http://dx.doi.org/10.1504/IJWET.2011.040725
12. Kalampokis, E., Tambouris, E., Tarabanis, K.: Ict tools for creating, expanding, and exploiting statistical linked open data, statistical. IAOS 33(2), 503–514 (2017)
13. Martin, M., Abicht, K., Stadler, C., Ngonga Ngomo, A.C., Soru, T., Auer, S.: Cubeviz: Exploration and visualization of statistical linked data. In: Proceedings of the 24th International Conference on World Wide Web. pp. 219–222. WWW '15 Companion, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2740908.2742848
14. Miles, A., Bechhofer, S.: SKOS simple knowledge organization system. Tech. rep., W3C (August 2009)
15. Oracle: Oracle olap developer's guide to the olap api, 10g release 2 (10.2) (2006)
16. Varga, J., Etcheverry, L., Vaisman, A.A., Romero, O., Pedersen, T.B., Thomsen, C.: Qb2olap: Enabling olap on statistical linked open data. In: 32nd International Conference on Data Engineering (ICDE). pp. 1346–1349. IEEE (May 2016)