VOLT: A Provenance-Producing, Transparent SPARQL Proxy for the On-Demand Computation of Linked Data and its Application to Spatiotemporally Dependent Data

Blake Regalia, Krzysztof Janowicz, and Song Gao

STKO Lab, University of California, Santa Barbara, USA
blake@geog.ucsb.edu, janowicz@ucsb.edu, sgao@geog.ucsb.edu

Abstract. Powered by Semantic Web technologies, the Linked Data paradigm aims at weaving a globally interconnected graph of raw data that transforms the ways we publish, retrieve, share, reuse, and integrate data from a variety of distributed and heterogeneous sources. In practice, however, this vision faces substantial challenges with respect to data quality, coverage, and longevity, the amount of background knowledge required to query distant data, the reproducibility of query results and their derived (scientific) findings, and the lack of computational capabilities required for many tasks. One key issue underlying these challenges is the trade-off between storing data and computing them. Intuitively, data that is derived from already stored data, changes frequently in space and time, or is the result of some workflow or procedure, should be computed. However, this functionality is not readily available on the Linked Data cloud with its current technology stack. In this work, we introduce a proxy that can transparently run on top of arbitrary SPARQL endpoints to enable the on-demand computation of Linked Data together with the provenance information required to understand how they were derived. While our work can be generalized to multiple domains, we focus on two geographic use cases to showcase the proxy's capabilities.

Keywords: Linked Data, Semantic Web, SPARQL, Geo-Data, Cyber-Infrastructure, Geospatial Semantics, VOLT

1 Introduction and Motivation

Linked Data described the paradigm for a Web of densely interconnected yet distributed data. It provided methods and tools that dramatically ease the publication, retrieval, sharing, reuse, and integration of semantically rich data across heterogeneous sources. Over the last few years, we have witnessed a rapid increase in available data sources on the Linked Data cloud and a fast uptake of the involved technologies in academia, governments, and industry. Nonetheless, several key issues remain to be addressed in order to enable the full potential of Linked Data. One of these issues is the trade-off between storing data and computing them. To give a concrete example, if the population and area of a

county are available, should the population density be stored as well or should it be computed on-demand as it depends on already stored properties? Storing such data is often problematic or even impossible for multiple reasons. Keeping the population density in sync with a changing population is just one example. Consequently, such statements should be computed. However, this functionality is not readily available on the Linked Data cloud and is not fully supported by existing query languages, endpoints, or APIs.

Recently, a variety of approaches [1, 4, 5, 9, 10] have been proposed to address this and related issues. Here, we argue why these approaches alone are not sufficient and propose a framework inspired by a combination of their findings. Essentially, we propose a proxy[1] that can transparently run on top of any SPARQL 1.1 compliant endpoint while providing a framework for the on-demand computation and caching of Linked Data. Going beyond existing work, our approach also provides the provenance information required to make sense of the (cached) results, thereby improving reproducibility. Essentially, all (derived) data together with the procedures used to compute them are stored as RDF in separate graphs.

In the following, and as space permits, we highlight key aspects of the VOLT[2] proxy and framework by example. Instead of focusing on technical (implementation) aspects alone, we showcase VOLT's capabilities by discussing two use cases in detail. These use cases also serve as the evaluation of our work, e.g., they demonstrate how to improve the data quality of DBpedia and reduce storage size at the same time. While our work can be generalized to multiple domains, both use cases focus on geo-data. We believe that the challenges introduced by spatiotemporal data are ideal for discussing the need for provenance information on the procedural (workflow) level, the difficulties resulting from keeping *dependent* data in sync, and the problem that allegedly *raw* data was created by using some latent assumptions that now hinder reproducibility and thus interoperability.

2 The VOLT Framework and Proxy

Work that aims at bringing API-like features to the Semantic Web typically does so by either suggesting ways to extend SPARQL or by providing additional functionality outside of the typical Semantic Web layer cake; see Section 4. Implementing such solutions often requires a custom SPARQL engine or the adoption of future W3C recommendations. Furthermore, running non-standard SPARQL engines threatens Linked Data interoperability and reusability of federated and non-federated queries alike. For these reasons, we often fail to see widespread use of experimental technologies. Finally, most of these technologies are not transparent, i.e., they require additional knowledge or at least awareness by the end user. To overcome these issues, we strive to develop a transparent framework that embraces the existing technology stack without any changes to SPARQL. Our approach functions as a *transparent proxy* [3] to any existing SPARQL 1.1

[1] A working VOLT proxy prototype is available at: `http://demo.volt-name.space/`
[2] VOLT: **V**OLT **O**ntology and **L**inked data **T**echnology

engine and thereby acts as a legitimate endpoint. When a query is issued to the proxy, it triggers a series of interactions with the underlying, encapsulated SPARQL endpoint before forwarding the results back to the client. In other words, the client does not notice any difference to a regular endpoint.

In this section we introduce the general VOLT architecture, highlight important aspects such as transparency, and give an overview of the implementation.

2.1 SPARQL as an API

The idea of using triples within the basic graph pattern of a query to invoke computation is referred to by iSPARQL as *virtual triples* [5]. A virtual triple uses the predicate to identify a procedure and effectively treats each subject and object of the triple as an input or output to the procedure. Like virtual triples and the *magic properties*[3] of Apache's ARQ, we use the triple's predicate as a way to identify a user-defined procedure. However, we make a distinction between the various ways in which these special patterns are used in our framework:

Firstly, *computable properties* simply represent an existential relation between two named entities. For example, consider a computable property named `udf:intersects` that tests for the spatial intersection between two individuals. A client may trigger computation on the individuals `:A` and `:B` by issuing a SPARQL ASK query with the basic graph pattern `:A udf:intersects :B`. Alternatively, a client may find all things that intersect with `:A` via a SELECT query `:A udf:intersects ?other`, where the object of the previous triple has been replaced by the variable `?other`. Yet another style allows the client to test multiple computable properties on the same triple by using a variable in place of the predicate along with a triple that constrains the variable to a specific `rdf:type`. For instance, a client may discover all topological relations between two particular regions by issuing the SELECT query `:A ?relation :B. ?relation a udf:RegionConnectionRelation`. In this variation, the triple that constrains the variable `?relation` functions as a *computable property trigger*. It indicates the client's intention to invoke testing on an entire class of computable properties.

Secondly, *functional triples* act as interfaces for calling user-defined procedures with named inputs and outputs. To invoke a procedure, functional triples expect a primary *root triple* where the subject is anonymous and the object is a blank node. The blank node object acts as a hashmap for both the input arguments and output variables to the procedure. Whereas EVT[4] functions accept an ordered list of input arguments and return a single RDF term, functional triples accept an unordered set of named input arguments, are capable of returning multiple output bindings, and allow both inputs and outputs to be either RDF terms or RDF graphs. Once the functional triple call is executed, the entire graph constructed within the blank node is saved (either temporarily or persistently) to a graph in the triplestore. Doing so enables auxiliary pattern groups within the same query to work as if the entire functional triple's blank node was matched

[3] https://jena.apache.org/documentation/query/extension.html#property-functions

[4] Extensible Value Testing

to an existing set of triples. The subject of a root triple must be an unbounded variable or a top-level blank node (that is, anonymous) in the query as the entire functional triple will be materialized and the subject will become a URI suffixed by a UUID[5]. A functional triple example will be shown in Listing 8.

Thirdly, *pattern rewriters* perform special expansions to the SPARQL query at runtime for patterns that may be otherwise impossible to write in a single query, such as subqueries that construct RDF graphs. A pattern rewriter is invoked by a functional triple which identifies the rewriter's procedure along with its input arguments. A group of query patterns gets associated to the rewriter by exploiting the GRAPH keyword in SPARQL. Consider an example where we want to select only the first valid object matched by a list of acceptable predicates that are semantically equivalent (in a certain context). Say we want to count the sum of populations given by the DBpedia `dbp:population` predicate for some distinct places. Since we do not want to count the same subject twice, we only want to match a single value for each subject. If a subject does not have a valid numeric literal belonging to the primary predicate `dbp:population`, then we opt for a secondary predicate, `dbp:populationTotal`. One can perform this in a regular SPARQL query as depicted in Listing 1.

```
select (sum(?population) as ?totalPopulation) where {
    {   ?s dbp:population ?population .
        filter(isNumeric(?population))
    } union {
        ?s dbp:populationTotal ?population .
        filter(isNumeric(?population))
        filter not exists {
            ?s dbp:population ?primary_population .
            filter(isNumeric(?primary_population))}}}
```

Listing 1 Select the sum of population counts using a preferred order of predicates in a query to a regular SPARQL endpoint.

As the number of predicates to test for increases, so does the number of `FILTER NOT EXISTS` blocks in each new `UNION` group. Furthermore, if we wanted to use a list of predicate IRIs from an RDF collection found in a triplestore, then this selection would be impossible to perform in a single query. Employing a pattern rewriter, we can automate building such queries in addition to having their bindings projected onto the surrounding query level; see Listing 2.

```
select (sum(?population) as ?totalPopulation) where {
    ?matcher volt:firstMatch [
        input:onVariable "?p"^^volt:Variable ;
        input:useValuesFrom (dbp:population dbp:populationTotal) ;
        input:sampleFromVariables ("?population"^^volt:Variable)  ] .
    graph ?matcher {
        ?s ?p ?population .
        filter(isNumeric(?population))   }}
```

Listing 2 The `?matcher` variable can be thought of as binding to the URI of a named, transient graph. In reality, the pattern rewriter's procedure will transform the patterns within the `GRAPH` group into a new subquery. This code snippet along with the expanded query can be seen in its entirety at `https://git.io/v2Nxb` .

[5] Universally Unique Identifier

2.2 Transparency and Reproducibility

A key limitation of previous approaches has been with the client's inability to inspect the source code behind an API function. Functions are not always trivial and their algorithms may overlook cornercases or depend on undocumented assumptions – leading to a breakdown in *semantic interoperability*. Our approach is to make the source code for all procedures readily accessible to the client by storing everything in the triplestore as RDF. Each procedure is serialized according to the VOLT ontology[6] and stored in the *model graph*. In order to execute a procedure, the proxy downloads a segment of the model graph and evaluates each step from the procedure's sequence of instructions. A simple example of an instruction is the assignment of a variable to an expression, e.g., `?x = ?y + ?z`, which applies the addition operator to the values stored in the variables `?y` and `?z`, then puts the result in the locally-scoped variable `?x`. In the model graph, this expression is serialized as an abstract syntax tree; shown in Listing 3.

```
... [ a volt:Assignment ;
    volt:name "?x"^^volt:Variable ;
    volt:gets [ a volt:BinaryOperation ;
        volt:operator "+"^^volt:Operator ;
        volt:lhs "?y"^^volt:Variable ;
        volt:rhs "?z"^^volt:Variable ]] ...
```

Listing 3 Abstract syntax tree of an assignment instruction for a VOLT procedure.

In taking this approach, we are able to statically evaluate the validity of a procedure's RDF serialization by using an ontology. Another example of a procedural instruction might be a SPARQL query, which has the benefit of referential integrity in its serialized form. This implies that a client can discover a procedure that depends on a particular IRI by querying the model graph for that IRI in the object position of a triple. E.g., we can discover any procedures that depend on the `geo:geometry` predicate by using the query shown in Listing 4.

```
describe ?procedure from named volt:graphs where {
    graph volt:graphs { ?modelGraph a volt:ModelGraph }
    graph ?modelGraph {
        ?procedure rdf:type/rdfs:subClassOf volt:Procedure .
        ?procedure (!</>)+ geo:geometry .  }}
```

Listing 4 Discover any VOLT procedures that depend on the `geo:geometry` predicate by using the nexus property path `(!</>)+`.

Thus, VOLT is transparent in two ways: (1) the proxy sits on top of a regular endpoint without a client noticing any difference, i.e., computed Linked Data behave as if they were stored in the underlying triplestore [3], and (2) procedures (defined by users or providers) are open for inspection.

2.3 Provenance

During execution of a procedure, all SPARQL queries and function calls are analyzed and recorded. Any information used during the evaluation of these transactions gets serialized as RDF triples and stored into a *provenance graph*. Those details are used to associate a cached triple to the inputs and expected outputs of

[6] https://github.com/blake-regalia/volt

SPARQL queries and function calls which led to that result. This offers two advantages: (1) the provenance of a cached triple is stored and remains available for inspection by which a client has the means to review source information that led a procedure to its conclusion and (2) it enables the invalidation of *stale cache*.

2.4 Caching and Cache Invalidation

To improve the performance of matching query patterns against computable properties and functional triples, we make use of caching. When caching is enabled by the proxy's host, each cacheable result is diverted to a persistent *output graph* instead of a temporary *results graph*. The input query is ultimately executed on the union of the source graph(s), results graph, and output graph, known collectively as the *content graph*. Determining whether or not a result should be cached depends on the ontological definition of the procedure that was used. Caching will only take place on a result when the procedure allows it. However, a client can bypass caching any results for the entire duration of a query's execution by including `optional {[] volt:ignoreCache true}` in the input query. Using the OPTIONAL keyword ensures that the query is reusable against arbitrary SPARQL engines, e.g., ones that do not run the proxy.

Each time a new triple is cached for a computable property, that triple runs the risk of being obsolete for future queries if the contents of its original source graph were to change. To detect this issue and protect against stale cache, we embed a cache invalidation feature within the framework. For procedures that use simple SPARQL queries, this may just involve confirming the existence of triples. In these cases, a cached result may be validated by a single query directed at the actual SPARQL engine. However, procedures that use more complex queries can employ patterns such as property paths or aggregate functions which can only be validated by executing those queries in full. We realize the need for an ontology that enables serializing, with various levels of complexity, methods of result validation for outputs of function calls and SPARQL queries given their inputs.

2.5 VOLT Procedures

The VOLT framework supports several types of user-defined procedures; each type serves a different purpose. In the model graph, a user may define procedures for EVT functions, computable properties, functional triples, and pattern rewriters. For each of these mediums, there is an ontological class that defines how an associated procedure must be encoded as RDF in the user-defined model graph. For example, a VOLT EVT function must have at least one member of type `volt:ReturnStage` in the RDF collection object pointed to by the `volt:stages` predicate within the procedure's set of defining triples.

The user-defined model exists as an RDF graph which encodes each procedure definition as a sequence of instructions. These instructions are limited to the basics, such as: operational expressions, control flow, SPARQL queries, and so on. To provide developers with the full flexibility of a programming language, the user-defined model can be extended by scripting plugins. Plugins are ideal for handling tasks such as complex calculations, networking, and I/O. They are

treated as namespaced modules. A single plugin may host an array of functions. For instance, we created a plugin that uses PostGIS[7] to handle geographic calculations; see Listing 5 for a call to the EVT function `postgis:azimuth`.

Under the hood, each plugin registers a specific namespace with the proxy by inserting RDF statements about itself into the model graph. This information includes metadata such as the path of the binary to execute, the path or URL of the source code if available, the namespace IRI, and process-related configuration. Anytime an EVT function call has a registered namespace, it will trigger the corresponding plugin. If the plugin is not already running, the proxy will load it into memory by spawning a child process. Once a plugin is running, the proxy pipes the function name and input arguments, serialized as JSON over stdin, to that child process. A single process may be used to run multiple tasks in series and multiple process of a plugin may be spawned in order to run tasks in parallel. Idle and busy processes may be terminated at the discretion of the proxy.

```
prefix postgis: <http://stko.geog.ucsb.edu/volt-plugins/postgis/#>
select ?angle where {
        dbr:Santa_Barbara geo:geometry ?wktFrom .
        dbr:Ventura geo:geometry ?wktTo .
        bind( postgis:azimuth(?wktFrom, ?wktTo) as ?angle ) }
```

Listing 5 Calls the user-defined EVT function 'azimuth' in the PostGIS plugin.

2.6 Query Flow Overview

To give a brief overview of how the proxy works, we examine VOLT's computable property feature. In Case Study I, we will demonstrate the use of such a property to determine the cardinal direction between Santa Barbara and Ventura. Figure 1 depicts the process of executing the procedure for `stko:east` as a flowchart.

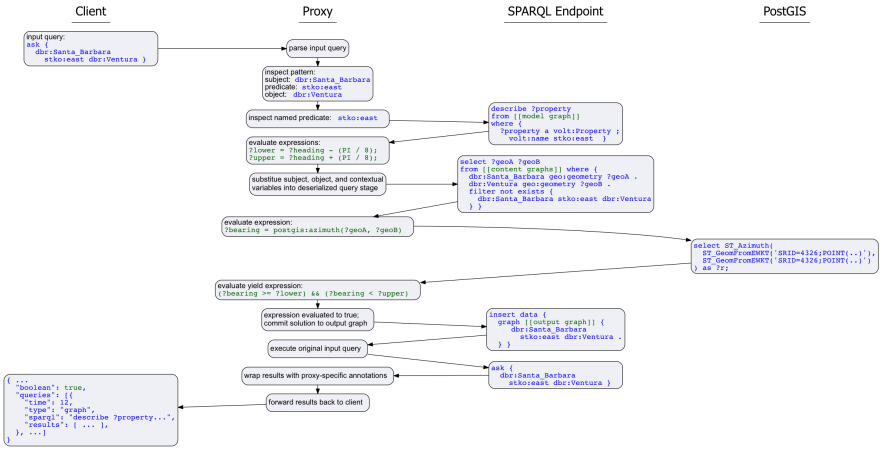

Fig. 1 The execution of computable property `stko:east` represented by a flowchart.

[7] http://postgis.net/

3 Case Studies

This section discusses two geographic use cases to showcase VOLT in action. Each use case highlights a different capability of the framework.

3.1 Case Study I: Cardinal Directions

The four cardinal directions North (N), South (S), East (E), and West (W) are among the most common means to express directional relations. The equal directional divisions of a compass rose are known as the four intercardinal directions, i.e., Northeast (NE), Southeast (SE), Southwest (SW) and Northwest (NW). In this section, *cardinal directions* will refer to all eight directions. Figure 2 shows how the bearing span ω for a cardinal direction is represented. The directionality is determined by testing if the azimuth between the point geometries of two places falls within ω from the primary angle of the direction. For the 8 cardinal directions, ω is set to $\pi/8$. For example, SE (*stko:southeast* here) covers the range $5\pi/8$ to $7\pi/8$ which is measured from the positive y-axis.

According to a SPARQL query for all resources of type `dbo:Place` or having a `geo:geometry`, there are over 1 million places in DBpedia.[8] Nearly 35,000 of them are associated to at least one triple with a cardinal direction predicate, leading to a total of 108,818 distinct triples involved. While this number is large, it is only a small portion ($\approx 1.2\%$) of the potential amount of cardinal direction relations among all places if merely storing a single triple per direction, e.g., only storing the nearest place to the North, South, and so forth. Trying to store all cardinal directions between all places would lead to a combinatorial explosion.

The entities contained in these triples vary widely and include macro-scale types such as *Mountain Range* or *Country*, meso-scale types, such as *City* or *River*, and micro-scale place types such as *Hospital*. Interestingly, types such as *Person* also show up, likely confusing persons with the places they were buried; e.g., `dbr:Saint Mechell is dbp:north of dbr:Tref Alaw`. Intuitively, and leaving cases such as headlands and meandering rivers aside, there should be only one cardinal direction relation between two places. Surprisingly, there are 3,411 places (involving $\approx$ 17,000 triples) with more than one cardinal direction to the same entity. For instance, Chicago, IL is both *dbp:east* and *dbp:west* of *Lincolnwood, Rosemont, and Schiller Park*, which is controversial. Consequently, we are compelled to test the accuracy of cardinal directions in DBpedia.

In order to compute the cardinal direction accuracy between entities of type `dbo:Place` that have one or more `geo:geometry` property in DBpedia, we selected all combinations of geometries between two places[9]. Our selection yielded 136,964 results of which 91,890 matched correctly, leaving 45,084 rows (33%) marked as incorrect. To validate that our computational representation of the cardinal directions does not introduce bias, we show in Figure 2 that each of the eight cardinal directions have roughly equal portions of incorrect relations. If we consider all 133,941 cardinal direction triples in DBpedia, we find that 55,928 (42%) of them have a subject or object lacking `geo:geometry`, or are not of type

[8] All queries & experiments were performed on the stable DBpedia 2015-04 version.
[9] Please note that some places have more than one geometry.

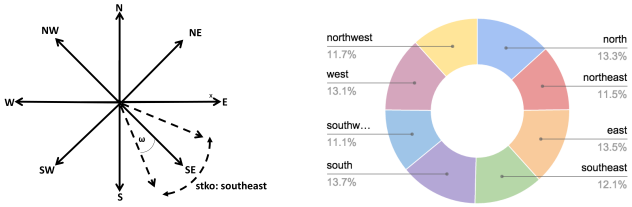

Fig. 2 The eight primary/inter-cardinal directions and their range (left) and the proportional distribution of mismatched directions normalized by categorical count (right).

`dbo:Place`. In fact, 17,957 triples have cardinal direction relations to RDF literals, 537 of which are of datatype `xsd:integer`. Most importantly, our argument is that given the few correct existing cardinal direction triples, a Linked Data user has to wonder why these specific relations are present in DBpedia and not a comprehensive set of cardinal directions between all places. This, however, would far exceed the total number of triples in DBpedia today. The imbalanced cardinal direction distribution becomes immediately clear by inspecting Table 1.

Table 1 Cardinal direction accuracies of the top 20 places with the most relations.

Place	Matches	Total	Accuracy	Place	Matches	Total	Accuracy
Wrexham	47	71	0.66	Karimnagar	27	38	0.71
Dolgellau	49	58	0.84	Ranchi	27	38	0.80
Ruthin	27	57	0.47	Shrewsbury	34	35	0.97
Bradford	29	53	0.55	Brothertoft	30	34	0.88
Bala, Gwynedd	22	47	0.47	Burton-upon-Trent	34	34	1.0
Orlando, Florida	27	47	0.57	Boston, Lincolnshire	25	33	0.76
Lichfield	43	46	0.93	Kirkby	29	33	0.88
Corwen	26	43	0.60	Ford, Shropshire	17	31	0.55
Aberystwyth	31	43	0.72	Mansfield	26	31	0.84
Derby	22	42	0.52	Glensanda	24	30	0.80

Another challenging issue is the computation of cardinal directions between polygonal representations of places. It is straightforward to compute point-to-point cardinal direction results on-the-fly if centroids are taken as the representations of regions. Depending on the polygons and the representativeness of centroids, there may be a varying degree of uncertainty associated with a cardinal direction relation between two regions. For example, according to DBpedia, the city of *Ventura* is linked to the city of *Santa Barbara* via the `dbp:northwest` relation, i.e., Ventura should be located to the *southeast* of Santa Barbara. This may be true for a certain point-feature representation of the cities but is not correct for all points inside the city boundaries. In fact, by taking the OpenStreetMap polygons for Santa Barbara and Ventura and defining a regular point grid of 1x1 km, we can compute the probability of grid points contained in Ventura to locate in the *southeast* of Santa Barbara (grid points). We filter out those points which are either outside of the city boundary or in the ocean. In total, we get 79 representative points for Santa Barbara and 88 points for Ventura. As depicted in Figure 3, we compute cardinal direction relations between 6,952 pairs of points in total. Our result shows that *southeast* is only the correct relation in 7.6% of the cases while it is *east* in 92.4% of the cases. That is to say that the

DBpedia statement of Ventura being southeast of Santa Barbara is merely true for 527 point pairs, while east is the correct relation for 6,425 other pairs. The situation would be even more complex if we consider fuzzy-set representation typically used for cognitive regions, e.g., *downtown*.

The last issue that remains to be discussed is performance. Clearly, computing cardinal directions takes longer than retrieving stored triples. A SPARQL query for all cardinal directions of the top 20 places takes about 3.3s on DBpedia's public endpoint. A *cold*, i.e., non-cached, VOLT prototype computes the same relations and returns its results (but does not yield erroneous data as does DBpedia) in about 18s on a modern laptop. This number should be taken into perspective by comparing it to the cache-enabled VOLT which takes only 6.9s after an initial run. Finally, it is important to remember that queries typically ask for the cardinal direction between a place and other geographic features and not for hundreds of directions among 20 random places. In such real-world cases, however, the overhead introduced by computation is relatively small.

Summing up, DBpedia currently only stores a very small, and from an end user's perspective, arbitrary fraction of cardinal direction relations. Approximately 33% of these relations are defective and many other need an understanding of the involved uncertainties to make use of them in a reproducible setting. For instance, there is no way for a user to understand what is returned by a SPARQL query for cardinal directions: are the results about the closest entity in a given direction, multiple entities, entities of the same type (e.g., the city north of LA), and so forth. Using the VOLT proxy, cardinal directions between all places can be computed on-demand along with provenance records that document how the computation was done and based on which formal definitions.

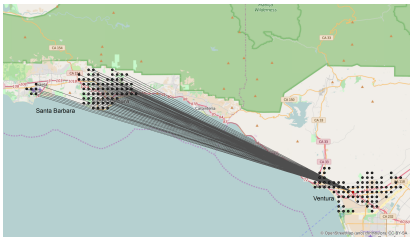

Fig. 3 Uncertainty in cardinal directions for Santa Barbara and Ventura.

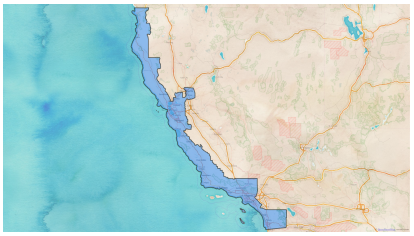

Fig. 4 Union of coastal counties computed as adjacent to Pacific Ocean.

3.2 Case Study II: Counting Regional Population

For the second case study, let us assume that a client wants to count the total population of California's coastal counties. She discovers the DBpedia resources for: North Coast of California, Central Coast of California and South Coast of California; each of which embodies counties along the coast. Intuitively, the user expects these three regions to be spatially disjoint and after inspecting the page for the Central Coast, naively devises the SPARQL query shown in Listing 6.

At the time of this writing, the query from Listing 6 returns a `?costalPopulation` of 2,249,558 - the same number as the population property

```
select (sum(?regionalPopulation) as ?coastalPopulation) where {
    ?region dbp:population ?regionalPopulation .
    values ?region {
        <http://dbpedia.org/resource/North_Coast_(California)>
        <http://dbpedia.org/resource/Central_Coast_(California)>
        <http://dbpedia.org/resource/South_Coast_(California)> }}
```

Listing 6 Select the sum of population counts for all three CA coastal regions.

given by the DBpedia resource for the Central Coast. In fact, the South Coast was not included since its population value is the literal "~ `20 million`" and the North Coast does not have a population property to begin with. Therefore, since the query does not check if each region was matched to a triple, and since the `sum` aggregate function in SPARQL *silently* ignores non-numeric values, the result of this query is misleading. Even more, the three coastal regions are neither continuous nor disjoint. For example, there are two coastal counties, San Francisco County and San Mateo County, which do not belong to any of the three coastal regions in California; they break the continuity of these regions by making a gap in between the North Coast and the Central Coast. The regions are also not disjoint because the Central Coast and the South Coast both include Ventura County; this could lead to counting the population of Ventura County twice.

Clearly, the client needs a better way to select the coastal counties of California and should be able to validate the accuracy of their operation by inspecting the provenance of constituent population values. By modifying our data to be GeoSPARQL-conformant, we can build a better query as shown in Listing 7.

```
# count the population of coastal counties in California
select (sum(?countyPopulation) as ?coastalPopulation) where {
    # get geometry of Pacific Coast as WKT
    data:PacificCoast geo:hasGeometry/geo:asWkt ?pacificCoastWkt .
    # use a subquery to group by place; avoid counting same place twice
    { select ?county (sample(?population) as ?countyPopulation) {
        # select all California counties and geometries as WKT
        ?county a yago:CaliforniaCounties .
        ?county geo:hasGeometry/geo:asWKT ?countyWkt .
        # make sure the county geometry is a polygon
        filter(regex(?countyWkt, '^(<[^>]*>)?(MULTI)?POLYGON', 'i'))
        # filter for coastal counties only
        filter(geof:sfTouches(?countyWkt, ?pacificCoastWkt))
        # get population of each county using best valid property name
        {   # best property to use is 'dbo:populationTotal'
            ?county dbo:populationTotal ?population .
            filter(isNumeric(?population))
        } union {
            # next best property is 'dbp:populationTotal'
            ?county dbp:populationTotal ?population .
            filter(isNumeric(?population))
            # block counties that have the preferred property
            filter not exists {
                ?county dbo:populationTotal ?best_population .
                filter(isNumeric(?best_population))
            }}} group by ?county }}
```

Listing 7 Use GeoSPARQL to count the population of California's coastal counties.

While the GeoSPARQL query is more likely to yield an accurate result, the user cannot perform aggregate spatial operations. In order to check if the entire coast was accounted for, she would have to issue a separate query in which `?countyWkt` is selected without any aggregate functions and then plot each

geometry on a map. With the VOLT framework however, we provide namespaced aggregate functions that construct temporary RDF graphs in the SPARQL query from a list of results for a single variable. By keeping only the county selection patterns in the subquery and aggregating those counties into an RDF Set, we can then call the user-defined `stko:sumOfPlaces` method to sum the values of the population properties as shown in Listing 8. Additionally, the user-defined method can construct a single geometry feature that is the union of all coastal counties in California. We then plot this geometry feature on a map to inspect the areas included in our population count, as shown in Figure 4.

```
prefix volt: <http://volt-name.space/ontology/>
prefix input: <http://volt-name.space/vocab/input#>
prefix output: <http://volt-name.space/vocab/output#>
prefix stko:  <http://stko.geog.ucsb.edu/vocab/>

# count the population of coastal counties in California
select ?population ?area where {
    # in a subquery, aggregate all California's coastal counties into a set
    { select (volt:cluster(?county) as ?setOfCounties) {
        # select only California counties
        ?county a yago:CaliforniaCounties .
        # ...that are 'along' the Pacific Coast (refers to a computable property)
        ?county stko:along data:PacificCoast . } }
    # let 'sumOfPlaces' method compute the total population of coastal counties
    [] stko:sumOfPlaces [
        input:places ?setOfCounties ;
        input:propertyList (dbo:populationTotal dbp:populationTotal) ;
        output:sum ?population ;
        output:coveredArea ?area ;  ]  }
```

Listing 8 Computes the sum of values for the first valid numeric property from `dbo:populationTotal` or `dbp:populationTotal` for all coastal counties in California.

The `stko:sumOfPlaces` method is stored in the model graph as RDF triples. To simplify the process of programming user-defined procedures in RDF, we developed the VOLT syntax and its compiler[6]. The language allows inline embedding of SPARQL query fragments, dynamically-scoped variables, operational expressions, and basic flow control. The VOLT source code for the `stko:sumOfPlaces` method is shown in Listing 9. At runtime, the population example will cause this method to generate the SPARQL query shown in Listing 10. Note that the VOLT language does not invalidate our claim of the proxy being transparent and only depending on well established W3C technologies. The language is only used to simplify the production of ontologically-compatible RDF statements which define custom functions and optionally their connections to external systems such as PostGIS. This language is not used for querying or any other functionality exposed to the client. As explained before, each procedure is serialized to RDF and stored in the model graph where it is available for public inspection.

```
method stko:sumOfPlaces {
    input ?places decluster into ?place
    input ?propertyList(list)
    select ?sum=sum(?value) ?placeGeomsWkt=volt:collect(?placeWkt) {
        ?matcher volt:firstMatch [
            input:forVariable "?property"^^volt:Variable ;
            input:useValuesFrom ?propertyList ;
            input:sampleFromVariables ("?value"^^volt:Variable)  ] .
        graph ?matcher {
            ?place ?property ?value .
            filter(isNumeric(?value)) }
```

```
        ?place geo:hasGeometry/geo:asWKT ?placeWkt }
    output ?sum  # shorthand for 'output [output:sum ?sum]'
    if object has output:placeGeometries {
        output [output:placeGeometries ?placeGeomsWkt] }
    if object has output:coveredArea {
        ?coveredArea = postgis:union(?placeGeomsWkt)
        output ?coveredArea }
    if object has output:overlap {
        ?overlap = postgis:union(postgis:intersectionAmong(?placeGeomsWkt))
        output ?overlap } }
```

Listing 9 User-defined `sumOfPlaces` method in VOLT syntax. It accepts two inputs: (1) a set of places whose properties should be summed and (2) a list of property IRIs ordered by the most preferred property value to match each distinct `?place`.

```
select (sum(?value) as ?sum)
    (group_concat(?_n3_placeWkt; separator='\n') as ?placeGeomsWkt)
where {
    # volt:firstMatch for variable ?property, use values from: (dbo:populationTotal
      ↪  dbp:populationTotal). sample from variable ?value
    { select ?place ?property (sample ?_sample_value as ?value)
        where {
            {   ?place ?property ?_sample_value .
                filter(isNumeric(?_sample_value))
                values ?property { dbo:populationTotal }
            } union {
                ?place ?property ?_sample_value .
                filter(isNumeric(?_sample_value))
                values ?property { dbp:populationTotal }
                filter not exists {
                    ?place dbo:populationTotal ?_0_value .
                    filter(isNumeric(?_0_value))
                }}} group by ?place ?property }
    ?place geo:hasGeometry/geo:asWKT ?placeWkt .
    # decluster ?places into ?place
    values ?place { dbr:Alameda_County   dbr:Contra_Costa_County  dbr:Del_Norte_County ... }
    # volt:collect(?placeWkt)
    bind( if(isBlank(?placeWkt), concat('_:', struuid()),
        if(isIri(?placeWkt), concat('<', str(?placeWkt), '>'),
        if(isLiteral(?placeWkt),
            concat('"',
                replace(
                    replace(str(?placeWkt), '"', '\\\\"'),
                    '\n', '\\\\n' ), '"',
                if(lang(?placeWkt) = '',
                    concat('^^<', str(datatype(?placeWkt)), '>'),
                    concat('@', lang(?placeWkt)) )),
        concat('?', struuid()) )))
        as ?_n3_placeWkt   ) }
```

Listing 10 A SPARQL query issued by the proxy on behalf of the client's input query. The client invokes the `stko:sumOfPlaces` method that substitutes values and subquery selection results into its own SELECT stage, ultimately yielding this SPARQL query.

Summing up, this second use case highlights the difficulties in naively querying Linked Data and the misleading results that commonly result from such queries. We use it to showcase VOLT's capabilities with respect to user (or provider) defined methods and the provenance information that allows others to inspect how the returned query results came to be.

4 Related Work

In this section we introduce work that is either related in terms of common goals, similar technological approaches, similar target domain, i.e., geo-data, or inspired and informed our thinking while developing the VOLT framework.

Linked Data Services (LIDS) [9] describes a formalization for connecting SPARQL queries to RESTful Web APIs by enabling a service layer behind query execution. Service calls have named inputs and outputs in the query. VOLT provides functional triples which also use named inputs and outputs in the query to make API calls to registered plugins. Plugins execute asynchronously and may perform networking tasks such as requests to RESTful Web APIs.

Linked Open Services (LOS) [7] sets forth the principles on how to establish interoperability between RESTful resources and Linked Open Data by semantically lifting flat content to RDF.

The **Linked Data API** (LDA) [8] is used to create RESTful APIs over RDF triple stores to streamline the process of web applications consuming Linked Data. Similar to LDA, VOLT also runs as a SPARQL proxy and dynamically generates SPARQL queries on behalf of the client.

Linked Data Fragments (LDF) [11] highlight the role of clients for scaling query engines by offloading partial execution to the web browser. Since our prototype is implemented in JavaScript, the proxy also runs as a standalone instance in the browser. The framework only needs a connection to a SPARQL endpoint over HTTP, or a locally emulated one such as the LDF client. In this regard, we aim to achieve Web-Scale querying as described by Verborgh [11].

SCRY [10] is a SPARQL endpoint that allows a client to invoke user-defined services by using named predicates in SPARQL queries. It simply identifies which service to execute and forwards the appropriate arguments given by the associated triple. SCRY's current implementation requires services to be implemented as Python modules or as command-line executables. Compared to VOLT, it does not provide the means for a client to inspect the source of user-defined services.

iSPARQL is a *virtual triple approach* [5] to invoking custom functions similar to the concept of *magic properties*. It extends the SPARQL grammar with a *SimilarityBlockPattern* production to distinguish between basic graph pattern triples and triple-like function calls having the form *?v apf:funct ArgList* [5].

The **SPIN** [1] framework generates entailments by issuing SPARQL queries to perform inferencing. The framework consists of a set of vocabularies that enable the serialization of user-defined rules, input as SPARQL queries, directly into an RDF graph; a technique that preserves IRI referential integrity. VOLT also serializes SPARQL fragments and graph patterns into RDF to use as inference rules. However, SPIN requires use of a proprietary extension of the SPARQL language to explicitly invoke computation while VOLT is designed to automatically recognize the need for computation on regular SPARQL queries that are issued as if the patterns are simply being matched to existing triples.

Logical Linked Data Compression [4] proposes a lossless compression technique which benefits large datasets when storage and sharing may be an issue. Similar to their compression, VOLT reduces the number of triples by using procedures to generate statements that can be deduced from source triples. However, our approach increases the total size of a dataset when caching is enabled. With caching disabled, one can instead opt for computing such statements on-demand thus saving storage space at the cost of query execution time.

5 Conclusions

In this work we introduced the transparent VOLT proxy for SPARQL endpoints. We outlined its core features, highlighted selected implementation details, and presented use cases that demonstrate the proxy's capabilities in addressing key shortcomings that we believe prevent the wide usage of Linked Data in science. Instead of storing triples that depend on already stored data, we propose to compute results on-demand and then cache them. Our work goes beyond merely reducing the amount of stored triples but also addresses quality issues as the *dependent* triples have to be kept in sync with their source data, e.g., when storing population densities in addition to population and areal data. We also address issues of provenance and the reproducibility of results by making the VOLT functions available and inspectable and by storing all data and procedures that were used to arrive at certain results in a separate graph. Finally, we discuss two use cases to demonstrate the difficulty in querying Linked Data, quality issues in Linked Data, and the need for the implemented VOLT capabilities.

Future work will focus on improving our current prototype and making it easier to extend and customize by others. We will also work on improving the proxy's performance and an alignment of our provenance and model graphs with ontologies such as PROV-O [6] and (semantic) workflow models in general [2].

Acknowledgements: This work was partially funded by NSF under award 1440202 and the USGS *Linked Data for the National Map* award. The authors would also like to thank Johannes Gross from NASA/JPL for his comments.

References

1. SPIN - SPARQL Inferencing Notation (2011). URL http://spinrdf.org/
2. Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., et al.: Examining the challenges of scientific workflows. IEEE computer **40**(12), 26–34 (2007)
3. Janowicz, K., Schade, S., Bröring, A., Keßler, C., Maué, P., Stasch, C.: Semantic enablement for spatial data infrastructures. Trans. in GIS **14**(2), 111–129 (2010)
4. Joshi, A.K., Hitzler, P., Dong, G.: Logical linked data compression. In: The Semantic Web: Semantics and Big Data, pp. 170–184. Springer (2013)
5. Kiefer, C., Bernstein, A., Stocker, M.: The Fundamentals of iSPARQL: A Virtual Triple Approach For Similarity-Based Semantic Web Tasks. Springer (2007)
6. Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., et al.: Prov-o: The prov ontology. W3C Recommendation **30** (2013)
7. Norton, B., Krummenacher, R.: Consuming dynamic linked data. In: COLD (2010)
8. Reynolds, D., Tennison, J., Dodds, L.: Linked Data API (2012). URL https://github.com/UKGovLD/linked-data-api
9. Speiser, S., Harth, A.: Integrating linked data and services with linked data services. In: Proceedings of the 8th Extended Semantic Web Conference (ESWC'11), pp. 170–184. Springer (2011)
10. Stringer, B., Meroño-Peñuela, A., Loizou, A., Abeln, S., Heringa, J.: To SCRY Linked Data: Extending SPARQL the Easy Way. In: Diversity++; ISWC 2015
11. Verborgh, R., Vander Sande, M., Colpaert, P., Coppens, S., Mannens, E., Van de Walle, R.: Web-scale querying through linked data fragments. In: Proceedings of the 7th Workshop on Linked Data on the Web (2014)