LSQ Framework: The LSQ Framework for SPARQL Query Log Processing

Claus Stadler[1,*], Muhammad Saleem[2] and Axel-Cyrille Ngonga Ngomo[2]

[1]*AKSW Research Group, University of Leipzig, Germany*
[2]*Data Science Group, Department of Computer Science, Paderborn University, Germany*

Abstract
The Linked SPARQL Queries (LSQ) datasets contain real-world SPARQL queries collected from the query logs of the publicly available SPARQL endpoints. In LSQ, each SPARQL query is represented as RDF with various structural and data-driven features attached. In this paper, we present the LSQ Java framework for creating rich knowledge graphs from SPARQL query logs. The framework is able to RDFize SPARQL query logs, which are available in different formats, in a scalable way. Furthermore, the framework offers a set of static and dynamic enrichers. Static enrichers derive information from the queries, such as their number of basic graph patterns and projected variables or even a full SPIN model. Dynamic enrichment involves additional resources. For instance, the benchmark enricher executes queries against a SPARQL endpoint and collects query execution times and result set sizes. This framework has already been used to convert query logs of 27 public SPARQL endpoints, representing 43.95 million executions of 11.56 million unique SPARQL queries. The LSQ queries have been used in many use cases such as benchmarking based on real-world SPARQL queries, SPARQL adoption, caching, query optimization, useability analysis, and meta-querying. Realization of LSQ required devising novel software components to (a) improve scalability of RDF data processing with the Apache Spark Big Data framework and (b) ease operations of complex RDF data models such as controlled skolemization. Following the spirit of OpenSource software development and the "don't repeat yourself" (DRY) paradigm, the work on the LSQ framework also resulted in contributions to Apache Jena in order to make these improvements readily available outside of the LSQ context.

Keywords
SPARQL, RDF, LSQ, Query Log, Software Framework

1. Introduction

Query logs allow us to bridge the theory and practice of SPARQL [1]. These query logs ensure that the research conducted by the community is guided by the requirements and trends that emerge in practice. The real-world SPARQL queries that are collected from public SPARQL endpoints have multiple use-cases such as performance evaluation in real-world settings, improve caching of triplestores, analysis on SPARQL adoption, query optimization, and usability analysis etc. [2]. The query logs produced by different SPARQL endpoints use different

6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs (QuWeDa) at ISWC 2022, virtual
*Corresponding author.
cstadler@informatik.uni-leipzig.de (C. Stadler); saleem@mail.uni-paderborn.de (M. Saleem); axel.ngonga@upb.de (A. N. Ngomo)
https://aksw.org/ClausStadler (C. Stadler)
0000-0001-9948-6458 (C. Stadler)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

formats to syntactically represent records. In this work we consider Web Log Format and CSV. Within each format, the set of available fields – the schemas – vary. For example, the formats used by wikidata, bio2rdf, and virtuoso instances in default configuration are different schema variants of the Web Log Format [2, 3]. Furthermore, in order to utilize real-world SPARQL queries in the aforementioned use-cases, it is required to parse these queries and annotate them with various structural and data-driven features such number of triple patterns, list of projection variables used, the different types of joins used, the results size etc. Finally, annotating SPARQL queries in existing query logs needs a scalable processing engine. For example, DBpedia public SPARQL endpoint receives more than 100k queries everyday. Similarly, the Wikidata public endpoint receives thousands of queries on daily bases.

To the best of our knowledge, there exist no generic and scalable software framework that parses these query logs to extract SPARQL queries, annotate them with different information, and convert them into an RDF dataset. To fill this research gap, we present the LSQ framework, which converts SPARQL query logs into RDF dataset and attach various structural and data-driven features to each SPARQL query. In order to perform scalable RDF conversion, we make use of the Apache Spark Big Data framework. By default, the framework supports query logs from nine different formats. Support for other logs formats can be easily done by adding log patterns into the configuration file.

This framework has already been used to convert query logs of 27 public SPARQL endpoints from various public SPARQL endpoints such as DBpedia, Wikidata, Bio2RDF, Semantic Web Dog Food etc. The resulting RDF datasets are named as LSQ V2.0 [2]. The LSQ V2.0 represents 43.95 million executions of 11.56 million unique SPARQL queries, resulting in 1.24 billion triples. The LSQ queries have been used in many use cases such as benchmarking based on real-world SPARQL queries [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], analysis of the SPARQL adoption in different applications [16, 17, 18], improving caching strategies for SPARQL engines [19, 20, 21, 22, 23], useability analysis of the SPARQL [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35], and SPARQL query optimization [36, 37, 38, 39]. Since the initial release of the LSQ V1.0 [3], the datasets converted by our framework have been used in more than 50 research papers[2]. The scalable components developed into the LSQ framework have also contributed to the recent Apache Jena 4.6.0 release with additional improvements pending[1].

The rest of the paper is organized as follows. In section 2, we present the LSQ framework. Section 3 discusses various use-cases where the datasets produced by our framework have already been used. In section 4, we present the useability instruction of this framework, followed by the availability information and sustainability plans in section 5. Finally, we conclude in section 6.

2. The LSQ Framework

In this section, we first briefly explain the core architecture. The details of each component is explained later.

[1]For details please see: https://github.com/apache/jena/pull/712, https://github.com/apache/jena/pull/1475, https://github.com/apache/jena/pull/1394, https://github.com/apache/jena/pull/1390, https://github.com/apache/jena/issues/1470, https://issues.apache.org/jira/projects/JENA/issues/JENA-2309

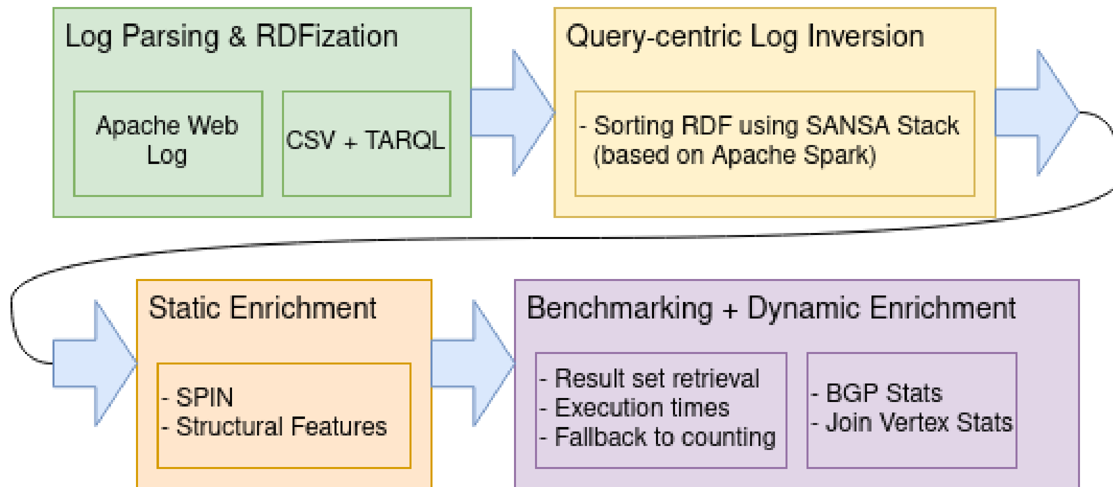

Figure 1: LSQ Framework Architecture. The underlying Semantic Web framework is Apache Jena.

Figure 1 shows the architecture of LSQ which is briefly summarized as follows. Records of query logs in different formats and having different schemas are parsed and normalized as RDF. The RDF log is inverted such that every query is related to all log records that mention it, i.e., every query is assigned a global (w.r.t, all query logs from different endpoints) identifier. In this process, queries are represented by hashes computed with their normalized strings. Static enrichment basically extracts the static features (that can be directly derived from a SPARQL query without running it against a specific RDF dataset) relevant to SPARQL query. It extends the RDF model with a SPIN[2] and further static structural features such as the number of projection variables, number of triple patterns, join types and selectivities, the used syntactic elements (e.g. UNION, OPTIONAL) etc. The dynamic enrichment extracts data-driven SPARQL features such as query runtime, result size or triple pattern selectivities etc. Whereas static enrichment is independent on any dataset, dynamic enrichment needs a dataset as input. Benchmarking is a form of dynamic enrichment that extends a query's RDF model w.r.t. a dataset with result set sizes and execution times. In the following, we present these steps in detail and how LSQ inspired creation of certain components which we contributed to other frameworks in order to facilitate reuse.

2.1. Named Graph Stream Processing

A fundamental design principle that is followed throughout LSQ is the *entity graph* paradigm, also referred to as (named-)graph-per-entity: An entity is represented by an IRI that appears in a (subject position of) triple in a named graph with the same IRI, such as `GRAPH :anEntity { :anEntity :p :o }`. Some Semantic Web tools (such as Virtuoso and Apache Jena) already support the use of named graphs with CONSTRUCT queries due to popular demand[3], although

[2]SPARQL SPIN representation: https://www.w3.org/Submission/spin-sparql/
[3]https://github.com/w3c/sparql-12/issues/31

this feature is not part of the current SPARQL specification v1.1[4]. LSQ uses the entity graph approach for representing log records and SPARQL queries. The advantage of this approach are:

- The graph name acts as a natural entry point for starting exploration/traversals of the contained data. This is a convention applications (such as viewers) can support without having to rely on a specific (ad-hoc) vocabulary.
- Retrieval and removal of all triples related to an entity is a simple operation on the named graph. This is of particular importance as certain models (e.g. SPIN) allow for arbitrary deeply nested tree structures in RDF which are extremely hard to query without scoping by a named graph.
- Partitioning the data into self-contained named graphs is well aligned with the map-reduce paradigm: Operations on individual entities, such as enrichment operations, can be carried out naturally in parallel over a set of entity graphs using conventional Big Data frameworks. We created the necessary hadoop-based parsers as part of LSQ and contributed them to the SANSA stack (see Section 2.7).
- An operation on an individual entity typically only requires its graph to be loaded rather than requiring access to *all* triples across all entities, which allows for stream-processing with low memory usage.

2.2. Parsing Query Logs

Generally, LSQ parses query logs and transforms them into a set of named graphs of which each represents an individual log record. This means that every log file is harmonized to a common RDF model.

Query logs come in different formats, and within a format there can be many different schemas. By schema we refer to the set of available attributes per log record. Manually determining the format and schema of a log is a very tedious task. For this reason, LSQ features a log format registry which can be used to probe log files against. Currently 2 types of log formats are supported: Web access log formats that are compatible with Apache HTTP server's mod_log_config[5] and CSV. For the former, LSQ provides a custom mapping of log fields to an RDF model, for the latter a tarql-based[6] approach is supported where columns of the input file are mapped to RDF using a SPARQL construct query. Support for additional mapping languages, such as RML [40] or YARRML [41] is future work. Note, that for the task of RDFizing a single CSV file, the practical difference between the approaches lies in syntax rather than functionality.

An excerpt of LSQ's log format registry configuration is shown in Listing 1. So far the default registry comprises 10 formats/schema combinations.

2.3. Accessing RDF graphs via Object Models

The LSQ data model (see Figure 2) is sufficiently complex such that manipulation solely with SPARQL turned out to be infeasible. One of the main reasons is due to the redundancy of

[4]https://www.w3.org/TR/sparql11-query/
[5]https://httpd.apache.org/docs/current/mod/mod_log_config.html
[6]https://github.com/tarql/tarql

Listing 1: RDF-based log format registry used in LSQ

```
fmt:combined
  a lsq:WebAccessLogFormat ;
  lsq:pattern "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\"" .

fmt:bio2rdfProcessedCsv
  a lsq:CsvLogFormat ;
  lsq:pattern
"""
PREFIX lsq: <http://lsq.aksw.org/vocab#>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
CONSTRUCT {
  GRAPH ?s {
    ?s
      lsq:query ?query ;
      lsq:host ?domain ;
      lsq:headers [ <http://example.org/header#User-agent> ?agent ] ;
      prov:atTime ?t
  }
} {
  BIND(IRI(CONCAT('urn:lsq:', MD5(CONCAT(?query, '-', ?domain, '-', ?timestamp)))) AS ?s)
  BIND(STRDT(?timestamp, xsd:dateTime) AS ?t)
}
""" .
```

complex graph patterns: Many patterns have to be repeated over and over again in different queries in order to address the nested resources. Especially during development, any change in the T-box requires a change in several places and debugging SPARQL queries that yield empty result sets is a tedious process. Furthermore, deterministic skolemization of the model is an important aspect: We strictly want to avoid effects such as a merge of a query's RDF resulting in doubling the number of related BGPs as would be the case if blank nodes were used. Therefore, we need a way to express how to compute the identity of entities, such as: the id of a triple pattern depends on the id of its subject, predicate and object components, the id of a BGP depends on the ids of the contained triple patterns (in order) and the id of a specific triple pattern in a BGP depends on that of the BGP and its own.

For this purpose, LSQ features a domain model realized as Java interfaces which must extend from Jena's `Resource` interface. The essence is, that the auto-generated implementations for this domain model then provides a *view* over the RDF graph; all state is kept in the RDF graph and any invocation of a "setter" method directly mutates the RDF graph whereas a "getter" method reads from it. Annotations on these interfaces are used to describe how getters and setters relate to triples in the underlying RDF graph. We use *Reprogen* (Resource Proxy Generator)[7] to generate Java proxies that implement the intended behavior of the annotations.

In order to realize skolemization, we extended Reprogen with new `@HashId` and `@StringId` annotations. @HashIds are a way to control how to compute hash codes for an RDF term in

[7]https://github.com/Scaseco/jenax/tree/develop/jenax-reprogen-parent/jenax-reprogen-core

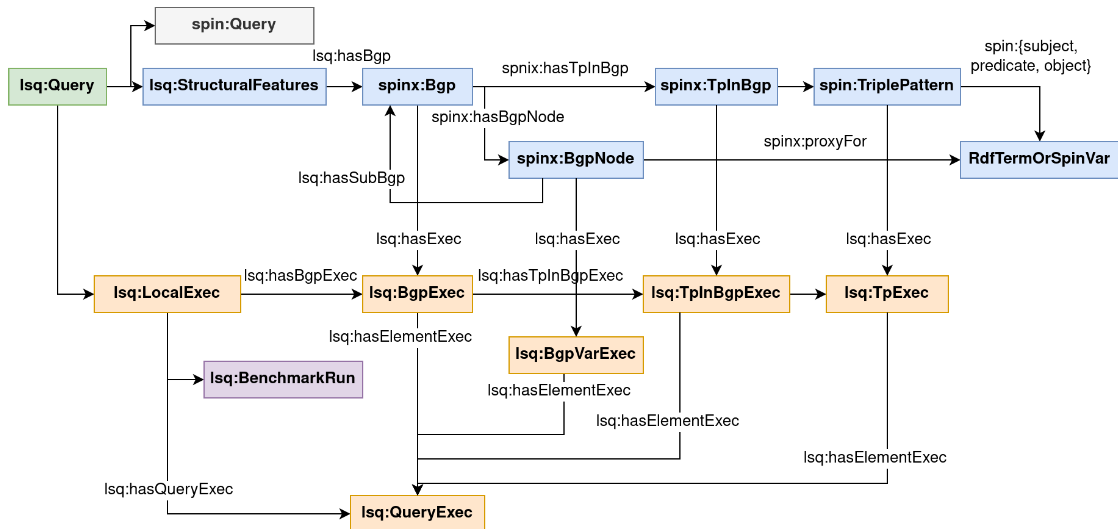

Figure 2: LSQ2 Data Model

an RDF graph w.r.t. to the annotated interface. The hash code of a resource is obtained by recursively considering the hash codes of all its member methods annotated with `@HashId`. The `@StringId` annotation is used for methods that can convert hash codes to strings suitable for use in IRIs. Listing 2 shows how to programmatically generate RDF for an annotated class whereas Listing 3 shows an example for how to annotate a class.

Listing 2: Example for setting up and skolemizing a basic LSQ RDF model

```
1 Model model = ModelFactory.createModel();
2 LsqTriplePattern tp = model.createResource(LsqTriplePattern.class);
3 TpInBgp tpInBgp = model.createResource(TpInBgp.class);
4 tp.setJenaTriple(new Triple());
5
6 Resource root = Reprogen.skolemize(tpInBgp, "http://lsq.aksw.org");
7 RDFDataMgr.write(System.out, root.getModel(), RDFFormat.TURTLE_PRETTY);
```

2.4. Query Id Generation

An important requirement in LSQ is determinism: Syntactically equivalent queries and their elements should always be represented by the same IRIs. The goal is to allow for the integration of different datasets generated by the same version of LSQ simply by means of merging the RDF. Generally, query strings are normalized using a parse/string serialization round trip. Query strings themselves are not suitable as identifiers because they can become very lengthy. The obvious choice is thus to resort to hashing. LSQ v1 used 8 characters of md5 hashes however it turned out that the number of queries was sufficiently large that collisions occurred. LSQ v2.0 uses sha256 with base64 encoding of the whole query string. The disadvantage was, that queries that only differed in projection or slice received unrelated hashes. As a consequence, identifying queries that only differ in their 'basic parameterization' was not possible from the hashes. LSQ

Listing 3: Example of LSQ's annotated Java domain model for which Reprogen generates proxy implementations that can read and write to an RDF graph

```
1  /* The identity of a bgp depends on the list of identities of the contained
2   * triple patterns */
3  public interface Bgp
4   extends Resource
5  {
6    @HashId
7    @Iri(LsqTerms.hasTp)
8    List<LsqTriplePattern> getTriplePatterns();
9  }
10
11 /* The identity of a "triple pattern in a bgp" depedends on the identity of
12  * the bgp and the identity of the triple pattern */
13 public interface TpInBgp
14   extends Resource
15 {
16     @HashId
17     @Iri(LsqTerms.hasBgp)
18     SpinBgp getBgp();
19     TpInBgp setBgp(Resource bgp);
20
21     @HashId
22     @Iri(LsqTerms.hasTp)
23     LsqTriplePattern getTriplePattern();
24     TpInBgp setTriplePattern(Resource tp);
25 }
```

v2.1 introduces additional structure: The *body hash* of a query is obtained by replacing the projection with 'SELECT *' and applying sha256+base64 hashing on the normalized query string. A separate hash is computed from the projection: The actual projection expression strings are first sorted lexicographically and a projection-base-hash is computed from these ordered strings. The actual projection is then permutation of the sorted one, and numbering schemes exist to label permutations: We use the *Lehmer-Code* to obtain a number for the actual projection. For a given sequence of n items, the Lehmer-Code is 0 when that sequence is sorted and the code reaches its maximum value $n!$ when the sequence is reverse-sorted.

Finally, the slice is appended which leads to the new improved pattern `bodyHash/projBaseHash/lehmerCode[/offset[-limit]]`.

2.5. LSQ Command Line Interface

In this section we briefly present the workflow for benchmarking a SPARQL endpoint with the `lsq` command line tool.

LSQ aims to capture on which endpoint a query was executed at which point in time. For that purpose, LSQ can reuse timestamps in log files. If such are absent then sequential numbering is used as a fallback. The URL of the endpoint from which a query log file originated needs to be provided manually. The endpoint for RDFization is purely of informational nature. No requests

will be made to it. In principle, if it was known which dataset was available at an endpoint at a certain point in time, then this information can be used to link to a dataset identifier. Furthermore, ideally a dataset identifier can be linked to a download URL of exactly the set of RDF triples (or quads) that were present at that endpoint. Although this background knowledge is currently usually not systematically available, we envision this situation to improve with advancements in data catalog and service modeling such as with DCAT[8]. An example of a command line invocation and its corresponding output is shown in Listing 4 and Listing 5.

Listing 4: Command for rdfizing a SPARQL query log

```
1 lsq rx rdfize --endpoint=http://dbpedia.org/sparql virtuoso.dbpedia.log
```

Listing 5: The output of the rdfization is one named graph per query

```
1  :lsqQuery-X {
2    :lsqQuery-X
3      lsq:text "SELECT * { ?s ?p ?o }" ;
4      lsq:hasRemoteExec :remoteExec-org-dbpedia-sparql_2016-04-10T01:00:00Z .
5
6    :remoteExec-org-dbpedia-sparql_2016-04-10T01:00:00Z
7      lsq:endpoint <http://dbpedia.org/sparql> ;
8      prov:atTime "2016-04-10T01:00:00Z"^^xsd:dateTime .
9  }
10 :lsqQuery-Y { ... }
```

Note, that the output is query-centric, which means that the input sequence of records is sorted by the query hash. The `rx` variant uses linux sorting which is not portable, whereas `spark` variant provides a portable java-native solution.

2.6. Benchmarking

LSQ uses RDF to keep benchmark settings in order to relate benchmark results to the context under which they were obtained. For this reason, benchmarking is a three step process: (1) Create a *benchmark configuration*. This is an RDF document which contains an IRI that carries the settings. The IRI is described with the date when the configuration was created and a custom label. The latter is useful for organization because benchmarking is often performed against SPARQL endpoints under a localhost URL from which no expressive name can be derived. Benchmark creation also caches the number of triples in the endpoint with the configuration. This number is used to compute certain ratios, such as the ratio of triples matched by a single triple pattern w.r.t. to the total size of the RDF graph. As a consequence, a new configuration should be created whenever the settings or the data changes. (2) Prepare a *benchmark run*: This is another little RDF document which only introduces an IRI with two pieces of information: The IRI of the configuration and the timestamp of when the run was prepared. (3) Execute the benchmark. Every benchmark task will be linked to the IRI of the benchmark run which in turn links to the used configuration. Listing 6 demonstrates the command line invocations for setting up and running a benchmark whereas an example configuration is shown in Listing 7.

[8] https://www.w3.org/TR/vocab-dcat-2/

Listing 6: Example benchmark setup and execution

```
lsq benchmark create --endpoint http://localhost:8080/sparql --dataset dbpedia
# Assumed output: xc-dbpedia_2021-10-22.conf.ttl
lsq benchmark prepare -c xc-dbpedia_2021-10-22.conf.ttl
# Assumed output: c-dbpedia_2021-10-22_2021-10-22T12_26_40_828056Z.run.ttl
lsq benchmark run -c xc-dbpedia_2021-10-22_2021-10-22T12_26_40_828056Z.run.ttl virtuoso.dbpedia.trig
```

Listing 7: Excerpt of benchmark configuration options

```
1  lsqr:xc-dbpedia_2021-10-22
2          dct:identifier                    "xc-dbpedia_2021-10-22" ;
3          lsqo:connectionTimeoutForRetrieval  "60"^^xsd:decimal ;
4          lsqo:executionTimeoutForRetrieval  "300"^^xsd:decimal ;
5          lsqo:maxResultCountForRetrieval  "1000000"^^xsd:long ;
6          lsqo:maxByteSizeForRetrieval    "-1"^^xsd:long ;
7          lsqo:maxResultCountForSerialization  "-1"^^xsd:long ;
8          lsqo:maxByteSizeForSerialization  "1000000"^^xsd:long ;
9          lsqo:executionTimeoutForCounting  "300"^^xsd:decimal ;
10         lsqo:connectionTimeoutForCounting  "60"^^xsd:decimal ;
11         lsqo:benchmarkSecondaryQueries  true .
```

2.6.1. Benchmark Workflow

When benchmarking a query, an IRI is allocated based on the benchmark run id and the query id. Within an individual benchmark run a query can only be executed once. A database (TDB2) is used to keep track of query hashes that have already been benchmarked. LSQ first attempts to benchmark and retrieve the result set of a query. If retrieval succeeds (and no thresholds are exceeded) then the retrieved result set is serialized as a literal in the output RDF dataset as long as the serialization thresholds are adhered to. If the retrieval fails due to threshold violation (in contrast to e.g. a syntactic error) then an alternate strategy is employed which attempts to count the size of the query by wrapping it with `SELECT (COUNT(*) AS ?c) { ... }`. The benchmark result contains triples for any exceeded thresholds.

Benchmarking Secondary Queries Primary queries are those originating from a query log. Secondary queries are those derived from the syntactic elements of primary ones. The most prominent elements are basic graph patterns and individual triple patterns. A secondary query is benchmarked just like a primary one. The rule that a query will only be benchmarked once within a run thus also applies.

An example RDF representation of a SPARQL query generated by the LSQ framework is shown in Listing 8. We encourage authors to have a look at LSQ V2.0 paper [2] for further details of the RDF representation.

Listing 8: An example LSQ/RDF representation of a SPARQL query in Turtle syntax [2]

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix lsqr: <http://lsq.aksw.org/> .
@prefix lsqv: <http://lsq.aksw.org/vocab#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix swc:  <http://data.semanticweb.org/ns/swc/ontology#> .
@prefix swr:  <http://data.semanticweb.org/> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix prov: <http://www.w3.org/ns/prov#> .

# Primary resource describing the query found with the SWDF logs
lsqr:lsqQuery-3wBd2uKotB_-vUxnngs6ZNsGPhJmIDD9c7ig0UI24y8
  lsqv:hasLocalExec lsqr:localExec-v9fBp3ElS1aVXXN1Z8zX1jxcHX3iy-axTgRrU2c7NY8 ;
  lsqv:hasRemoteExec lsqr:re-data.semanticweb.org-sparql_2014-05-22T16:08:17Z ,
    lsqr:re-data.semanticweb.org-sparql_2014-05-20T13:24:13Z ;
  lsqv:hasStructuralFeatures lsqr:lsqQuery-3wBd2uKotB_-vUxnngs6ZNsGPhJmIDD9c7ig0UI24y8-sf ;
  lsqv:hash "3wBd2uKotB_-vUxnngs6ZNsGPhJmIDD9c7ig0UI24y8" ;
  lsqv:text """PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
               PREFIX  swc:  <http://data.semanticweb.org/ns/swc/ontology#>
               SELECT DISTINCT  ?prop
               WHERE { ?obj  rdf:type swc:SessionEvent ; ?prop ?targetObj FILTER isLiteral(?targetObj)  }
               LIMIT   150""" .

# Static features of the query
lsqr:lsqQuery-3wBd2uKotB_-vUxnngs6ZNsGPhJmIDD9c7ig0UI24y8-sf
  lsqv:bgpCount 1 ;
  lsqv:hasBgp lsqr:bgp-_x9Mckke-V9R3ddISuw-Nj_j278nT5HwiA1WUNk7tgY ;
  lsqv:joinVertexCount 1 ;
  lsqv:joinVertexDegreeMean 2 ;
  lsqv:joinVertexDegreeMedian 2 ;
  lsqv:projectVarCount 1 ;
  lsqv:tpCount 2 ;
  lsqv:tpInBgpCountMax 2 ;
  lsqv:tpInBgpCountMean 2 ;
  lsqv:tpInBgpCountMedian 2 ;
  lsqv:tpInBgpCountMin 2 ;
  lsqv:usesFeature lsqv:fn-isLiteral , lsqv:Select , lsqv:Limit , lsqv:Functions , lsqv:Group , lsqv:Filter ,
    lsqv:Distinct , lsqv:TriplePattern .

# Remote execution no. 1 on the original endpoint
lsqr:re-data.semanticweb.org-sparql_2014-05-22T16:08:17Z
  prov:atTime "2014-05-22T16:08:17Z"^^xsd:dateTime ;
  lsqv:endpoint swr:sparql ;
  lsqv:hostHash "O5UQpDtofxAsrJk7yzGfDolFGylMFw5446KcRZDcBkU" .

# Remote execution no. 2 on the original endpoint
lsqr:re-data.semanticweb.org-sparql_2014-05-20T13:24:13Z
  prov:atTime "2014-05-20T13:24:13Z"^^xsd:dateTime ;
  lsqv:endpoint swr:sparql ;
  lsqv:hostHash "7aPNvqsgizRuEjH7_cO_dXoqLk-exKJ-xFmbCH3ew_E" .

# Local execution to extract statistics
lsqr:localExec-v9fBp3ElS1aVXXN1Z8zX1jxcHX3iy-axTgRrU2c7NY8-xc
  lsqv:benchmarkRun lsqr:xc-swdf_2020-09-23_at_23-09-2020_17:10:19 ;
  lsqv:hasQueryExec lsqr:queryExec-Cmv7SccybbBxwkep_cHvDiF3piq29tH7NWlDfIiCHqU .

# Results of local execution
lsqr:queryExec-Cmv7SccybbBxwkep_cHvDiF3piq29tH7NWlDfIiCHqU
  prov:atTime "2020-09-23T15:27:36.325Z"^^xsd:dateTime ;
  lsqv:countingDuration 0.008466651 ;
  lsqv:evalDuration 0.008868635 ;
  lsqv:resultCount 16 .

# The full data further include a SPIN description of the query, a list of BGPs within the query,
# a list of triple patterns and terms within the query, as well as execution statistics for individual
# BGPs, triple patterns and sub-BGPs induced by join variables
```

2.7. Scaling LSQ with SANSA

Processing large log files with only a single core is tedious and anachronistic in times of Big Data and laptops having more than a dozen cores. Apache Spark is a framework that enables scaling computing tasks to use all available resources on a cluster – even if the "cluster" only comprises a single machine. Apache Spark features high level abstractions for distributed executions of operations on different types of distributed collections of records. *Resilient Distributed Datasets* (RDDs) are the ones used by LSQ/Spark. However, the low-level I/O for reading records from files (regardless whether in local or distributed file systems) is provided by Apache Hadoop. The now retired Apache Jena/Elephas[9] project provided distributed ingestion of RDF data by wiring its own I/O library (called *RIOT*) up with Apache Hadoop. However, while Elephas supported n-quads, this format is both much more verbose and more hard to read than pretty printed trig. Conversely, manually reviewing rather sophisticated LSQ models in trig format was significantly easier, however, Elephas could not read trig in splits. Therefore use of this format negated the benefits of the Big Data framework. In order to optimize processing, we created an initial distributed parser for trig that searched hadoop input splits by matching the `<graphname> { ... }` pattern. This framework was continuously extended to handle RDF prefixes and even RDF data in literals. By now, this parsing framework has evolved into *Hadoop Generic Parser Framework* (HGPF) and provides support for trig (a superset of turtle, n-quads and n-triples), JSON and CSV.

Listing 9 shows the contribution made to SANSA in order to enable processing of large trig files (of which n-quads is a special case) with LSQ. The parser has been successfully used to ingest and sort 500GB of LSQ trig data in about 5 hours on a 3 node spark cluster.

Listing 9: Parsing named graph streams

```
1 RdfSourceFactory rdfSourceFactory = RdfSourceFactoryImpl.from(sparkSession);
2 RdfSource rdfSource = rdfSourceFactory.get("query-log.lsq.trig");
3 RDD<Quad> rddOfQuads = rdfSource.asQuads();
4 RDD<DatasetOneNg> rddOfDataset = rdfSource.asDatasets();
```

The HGPF framework has also been used to realize a CSV parser that can also handle multi-line cell values. The limitation is that when searching for candidate record offsets in a split, the maximum size of multi-line cells needs to be configured in advance. The default value is 500KB, and the candidate record offset detector always has to exhaust this amount of data in order not to miss any cell endings.

The CSV settings are based on the *frictionless data csv dialect* which slightly extends over the *CSV on the Web* (CSVW)[10] specification. An example of programmatic usage is shown in Listing 10.

Listing 10: Example for setting up and skolemizing a basic LSQ RDF model

```
1 JavaRDD<Binding> rddOfBindings = CsvDataSources.createRddOfBindings(sparkContext, "data.csv", csvDialect);
2 Query query = QueryFactory.create("CONSTRUCT ...");
3 JavaRDD<Quad> rddOfQuads = JavaRddOfBindingsOps.tarqlQuads(rddOfBindings, query);
```

[9] https://jena.apache.org/documentation/archive/hadoop/
[10] https://www.w3.org/TR/tabular-data-model/

3. Impact

To the best of our knowledge, the framework we propose in this paper is the first generic framework for RDFizing SPARQL query logs in different formats. Furthermore, it does so in a scalable way by following Big Data processing paradigms. There was no mechanism available to reuse existing query logs in a single standard format with much more enriched information attached to each query. Our framework is completely abided by semantic web technologies. It has been used to convert query logs of 27 public SPARQL endpoints, resulting in terabytes of RDF data.

In the recent LSQ v2.0 paper [2], potential six use cases – custom benchmarking, SPARQL adoption, caching, usability analysis, query optimisation, and meta-querying – have been discussed. The RDF datasets generated using the LSQ framework have been used widely for these use-cases [2]. he study [2] reported that 29 research papers have used LSQ queries for custom benchmarking, six research papers for SPARQL adoption, five research papers for caching, 12 research papers for SPARQL useability analysis, seven research papers for query optimisation, and two papers for meta-querying [2]. Furthermore, [2] discussed a number of works which have used LSQ (mostly for evaluation) in contexts that were not originally anticipated by the aforementioned use cases. These works include predicting temporal relations between events [42], augmenting RDF data sources with completeness statements [43], finding the frequency and distribution of answerable and non-answerable query patterns [43], question answering over linked data [44], and a blockchain that allows users to propose updates to faulty or outdated data [45].

4. Reusability

We are hopeful that the LSQ framework will be used by more researchers to convert existing query logs and create LSQ datasets. The resource home page includes documentation along with examples for easy reusability. The components developed in the LSQ framework are very generic. A dockerized version of the LSQ is also available from the resource homepage. The LSQ framework can also be adopted to other log formats by providing the log pattern, e.g., CSV, specific text pattern. Custom enrichment can also be done, i.e, add more query features attached to each query. The resource homepage includes a wiki explaining how others can use the framework along with examples and CLI instructions. So far, we have not tested the framework with XML, JSON logs. However, it should be working provided that the correct log pattern is specified in the configuration file.

5. Availability and Sustainability

The resource is available from a persistent URL http://w3id.org/lsq. LSQ v2.0 [2] is the canonical citation associated with this resource. Our framework and LSQ datasets are available under GNU General Public License v3.0. The source code is publicly available via Github. All future extensions will be reflected on the same persistent URL. In addition, this framework will be sustained via the Paderborn Center for Parallel Computing PC^2, which provides computing

resources as well as consulting regarding their usage to research projects at Paderborn University and also to external research groups. The Information and Media Technologies Centre (IMT) at Paderborn University also provides a permanent IT infrastructure to host the LSQ project.

6. Conclusion

In this paper, we have presented the LSQ framework, a scalable engine to represent queries in logs as RDF, allowing users perform their analysis on real-world SPARQL queries. We discussed the core architecture of this software framework along with potential impact and the useability instructions. We briefly discussed various use-cases where RDF datasets produced by our framework have already been used. In the future, we want to further extend this framework that provides further annotated information, e.g. annotating the named entities used in the SPARQL queries and their disambiguation to well-known datasets such Wikidata and DBpedia. We aim to collect further query logs from public SPARQL endpoints and provide them as RDF datasets.

Resource Availability Statement:

- Source code of the LSQ framework is available from https://github.com/AKSW/LSQ
- Installation instructions available from http://lsq.aksw.org/v2/setup.html
- Usage instructions available from http://lsq.aksw.org/v2/usage/usage.html
- RDF dumps of the LSQ v2.0 datasets available from https://hobbitdata.informatik.uni-leipzig.de/lsqv2/dumps/
- LSQ V2.0 public SPARQL endpoint is available from http://lsq.aksw.org/sparql
- Set of useful SPARQL queries over LSQ datasets available from http://lsq.aksw.org/v2/usage/usage.html
- The resource type is Software Framework and is available under GNU General Public License v3.0
- All of the above information along with legacy data and old LSQ maintenance information available from the resource persistent URL http://w3id.org/lsq
- LSQ V2.0 [2] is the canonical citation associated with this paper

Acknowledgments

The authors also acknowldge the financial support for 3DFed (Grant no. 01QE2114B), Know-Graphs(Grant no. 860801) and by the Federal Ministry for Economics and Climate Action in the project CoyPu (project number 01MK21007A).

References

[1] W. Martens, T. Trautner, Bridging Theory and Practice with Query Log Analysis, SIGMOD Record 48 (2019) 6–13.

[2] C. Stadler, M. Saleem, Q. Mehmoodb, C. Buil-Arandac, M. Dumontierd, A. Hogane, A.-C. N. Ngomoa, Lsq 2.0: A linked dataset of sparql query logs, in: Semantic Web Journal, 2022.
[3] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, A. N. Ngomo, LSQ: The Linked SPARQL Queries Dataset, in: International Semantic Web Conference (ISWC), Springer, 2015, pp. 261–269.
[4] M. Saleem, Q. Mehmood, A. N. Ngomo, FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework, in: International Semantic Web Conference (ISWC), Springer, 2015, pp. 52–69.
[5] M. Saleem, Q. Mehmood, C. Stadler, J. Lehmann, A. N. Ngomo, Generating SPARQL Query Containment Benchmarks Using the SQCFramework, in: ISWC Posters & Demos, CEUR-WS.org, 2018.
[6] M. Saleem, A. Hasnain, A.-C. N. Ngomo, LargeRDFBench: A billion triples benchmark for SPARQL endpoint federation, Journal of Web Semantics 48 (2018) 85–125.
[7] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, A. N. Ngomo, How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks, in: World Wide Web Conference (WWW), ACM, 2019, pp. 1623–1633.
[8] D. Hernández, A. Hogan, C. Riveros, C. Rojas, E. Zerega, Querying Wikidata: Comparing SPARQL, Relational and Graph Databases, in: International Semantic Web Conference (ISWC), Springer, 2016, pp. 88–103.
[9] J. D. Fernández, J. Umbrich, A. Polleres, M. Knuth, Evaluating query and storage strategies for RDF archives, Semantic Web 10 (2019) 247–291.
[10] A. Azzam, J. D. Fernández, M. Acosta, M. Beno, A. Polleres, SMART-KG: Hybrid Shipping for SPARQL Querying on the Web, in: The Web Conference (WWW), 2020, pp. 984–994.
[11] A. Bigerl, F. Conrads, C. Behning, M. A. Sherif, M. Saleem, A.-C. Ngonga Ngomo, Tentris – A Tensor-Based Triple Store, in: International Semantic Web Conference (ISWC), Springer, 2020, pp. 56–73.
[12] A. Azzam, C. Aebeloe, G. Montoya, I. Keles, A. Polleres, K. Hose, WiseKG: Balanced Access to Web Knowledge Graphs, in: The Web Conference (WWW), ACM / IW3C2, 2021, pp. 1422–1434. URL: https://doi.org/10.1145/3442381.3449911. doi:10.1145/3442381.3449911.
[13] A. Davoudian, L. Chen, H. Tu, M. Liu, A Workload-Adaptive Streaming Partitioner for Distributed Graph Stores, Data Science and Engineering 6 (2021) 163–179.
[14] A. A. Desouki, F. Conrads, M. Röder, A.-C. N. Ngomo, SYNTHG: Mimicking RDF Graphs Using Tensor Factorization, in: International Conference on Semantic Computing (ICSC), 2021, pp. 76–79. doi:10.1109/ICSC50631.2021.00017.
[15] M. Röder, P. T. S. Nguyen, F. Conrads, A. A. M. da Silva, A.-C. N. Ngomo, Lemming – Example-based Mimicking of Knowledge Graphs, in: International Conference on Semantic Computing (ICSC), 2021, pp. 62–69. doi:10.1109/ICSC50631.2021.00015.
[16] X. Han, Z. Feng, X. Zhang, X. Wang, G. Rao, S. Jiang, On the statistical analysis of practical SPARQL queries, in: International Workshop on Web and Databases (WebDB), ACM, 2016, p. 2.
[17] A. Bonifati, W. Martens, T. Timm, An Analytical Study of Large SPARQL Query Logs, PVLDB 11 (2017) 149–161. URL: http://www.vldb.org/pvldb/vol11/p149-bonifati.pdf. doi:10.14778/3149193.3149196.

[18] A. Bonifati, W. Martens, T. Timm, DARQL: Deep Analysis of SPARQL Queries, in: WWW Posters & Demos, ACM, 2018, pp. 187–190.
[19] M. Knuth, O. Hartig, H. Sack, Scheduling refresh queries for keeping results from a SPARQL endpoint up-to-date, in: On The Move to Meaningful Internet Systems (OTM), Springer, 2016, pp. 780–791.
[20] U. Akhtar, M. A. Razzaq, U. U. Rehman, M. B. Amin, W. A. Khan, E.-N. Huh, S. Lee, Change-Aware Scheduling for Effectively Updating Linked Open Data Caches, IEEE Access 6 (2018) 65862–65873.
[21] U. Akhtar, A. Sant'Anna, S. Lee, A Dynamic, Cost-Aware, Optimized Maintenance Policy for Interactive Exploration of Linked Data, Applied Sciences 9 (2019) 4818.
[22] J. Salas, A. Hogan, Canonicalisation of Monotone SPARQL Queries, in: International Semantic Web Conference (ISWC), Springer, 2018, pp. 600–616.
[23] T. Safavi, C. Belth, L. Faber, D. Mottin, E. Müller, D. Koutra, Personalized knowledge graph summarization: From the cloud to your pocket, in: International Conference on Data Mining (ICDM), IEEE, 2019, pp. 528–537.
[24] M. Arenas, G. I. Diaz, E. V. Kostylev, Reverse engineering SPARQL queries, in: World Wide Web Conference (WWW), ACM, 2016, pp. 239–249.
[25] F. Benedetti, S. Bergamaschi, A model for visual building SPARQL queries, in: Symposium on Advanced Database Systems (SEBD), 2016, pp. 19–30.
[26] I. Dellal, S. Jean, A. Hadjali, B. Chardin, M. Baron, On addressing the empty answer problem in uncertain knowledge bases, in: International Conference on Database and Expert Systems Applications (DEXA), Springer, 2017, pp. 120–129.
[27] T. Stegemann, J. Ziegler, Investigating learnability, user performance, and preferences of the path query language SemwidgQL compared to SPARQL, in: International Semantic Web Conference (ISWC), Springer, 2017, pp. 611–627.
[28] A. Viswanathan, G. de Mel, J. A. Hendler, Feature-based reformulation of entities in triple pattern queries, CoRR abs/1807.01801 (2018). URL: http://arxiv.org/abs/1807.01801.
[29] J. Potoniec, Learning SPARQL Queries from Expected Results, Computing and Informatics 38 (2019) 679–700.
[30] M. Wang, J. Liu, B. Wei, S. Yao, H. Zeng, L. Shi, Answering why-not questions on SPARQL queries, Knowledge and Information Systems (2019) 1–40.
[31] A. Bonifati, W. Martens, T. Timm, An analytical study of large SPARQL query logs, VLDB J. 29 (2020) 655–679. doi:10.1007/s00778-019-00558-9.
[32] X. Jian, Y. Wang, X. Lei, L. Zheng, L. Chen, SPARQL Rewriting: Towards Desired Results, in: SIGMOD International Conference on Management of Data, 2020, pp. 1979–1993.
[33] X. Zhang, M. Wang, M. Saleem, A.-C. N. Ngomo, G. Qi, H. Wang, Revealing Secrets in SPARQL Session Level, in: International Semantic Web Conference (ISWC), Springer, 2020, pp. 672–690.
[34] J. M. Almendros-Jiménez, A. Becerra-Terón, Discovery and diagnosis of wrong SPARQL queries with ontology and constraint reasoning, Expert Systems with Applications 165 (2021) 113772. URL: https://www.sciencedirect.com/science/article/pii/S0957417420305960. doi:https://doi.org/10.1016/j.eswa.2020.113772.
[35] M. Wang, K. Chen, G. Xiao, X. Zhang, H. Chen, S. Wang, Explaining similarity for SPARQL queries, World Wide Web (2021) 1–23.

[36] Z. Song, Z. Feng, X. Zhang, X. Wang, G. Rao, Efficient approximation of well-designed SPARQL queries, in: International Conference on Web-Age Information Management (WAIM), Springer, 2016, pp. 315–327.
[37] W. Martens, T. Trautner, Evaluation and Enumeration Problems for Regular Path Queries, in: International Conference on Database Theory (ICDT), Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018, pp. 19:1–19:21.
[38] S. Cheng, O. Hartig, OPT+: A Monotonic Alternative to OPTIONAL in SPARQL, Journal of Web Engineering 18 (2019) 169–206.
[39] D. Figueira, A. Godbole, S. N. Krishna, W. Martens, M. Niewerth, T. Trautner, Containment of simple conjunctive regular path queries, in: International Conference on Principles of Knowledge Representation and Reasoning (KR), 2020, pp. 371–380. doi:10.24963/kr.2020/38.
[40] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, Rml: a generic language for integrated rdf mappings of heterogeneous data, in: Ldow, 2014.
[41] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative Rules for Linked Data Generation at your Fingertips!, in: Proceedings of the 15th ESWC: Posters and Demos, 2018.
[42] K. Georgala, M. A. Sherif, A.-C. N. Ngomo, An efficient approach for the generation of Allen relations, in: European Conference on Artificial Intelligence (ECAI), IOS Press, 2016, pp. 948–956.
[43] P. Fafalios, Y. Tzitzikas, How many and what types of SPARQL queries can be answered through zero-knowledge link traversal?, in: ACM/SIGAPP Symposium on Applied Computing (SAC), ACM, 2019, pp. 2267–2274.
[44] K. Singh, M. Saleem, A. Nadgeri, F. Conrads, J. Z. Pan, A.-C. N. Ngomo, J. Lehmann, Qaldgen: Towards microbenchmarking of question answering systems over knowledge graphs, in: International Semantic Web Conference (ISWC), Springer, 2019, pp. 277–292.
[45] C. Aebeloe, G. Montoya, K. Hose, ColChain: Collaborative Linked Data Networks, in: The Web Conference (WWW), ACM / IW3C2, 2021, pp. 1385–1396. URL: https://doi.org/10.1145/3442381.3450037. doi:10.1145/3442381.3450037.