A Performance Study of RDF Stores for Linked Sensor Data

Hoan Nguyen Mau Quoc [a,*], Martin Serrano [b], Han Nguyen Mau [c], John G. Breslin [d] , Danh Le Phuoc [e]
[a] *Insight Centre for Data Analytics, National University of Ireland Galway, Ireland*
E-mail: hoan.quoc@insight-centre.org
[b] *Insight Centre for Data Analytics, National University of Ireland Galway, Ireland*
E-mail: martin.serrano@insight-centre.org
[c] *Information Technology Department, Hue University, Viet Nam*
E-mail: nmhan@hueuni.edu.vn
[d] *Confirm Centre for Smart Manufacturing and Insight Centre for Data Analytics, National University of Ireland Galway, Ireland*
E-mail: john.breslin@nuigalway.ie
[e] *Open Distributed Systems, Technical University of Berlin, Germany*
E-mail: danh.lephuoc@tu-berlin.de

Editors: First Editor, University or Company name, Country; Second Editor, University or Company name, Country
Solicited reviews: First Solicited Reviewer, University or Company name, Country; Second Solicited Reviewer, University or Company name, Country
Open reviews: First Open Reviewer, University or Company name, Country; Second Open Reviewer, University or Company name, Country

Abstract. The ever-increasing amount of Internet of Things (IoT) data emanating from sensor and mobile devices is creating new capabilities and unprecedented economic opportunity for individuals, organisations and states. In comparison with traditional data sources, and in combination with other useful information sources, the data generated by sensors is also providing a meaningful spatio-temporal context. This spatio-temporal correlation feature turns the sensor data become even more valuables, especially for applications and services in Smart City, Smart Health-Care, Industry 4.0, etc. However, due to the heterogeneity and diversity of these data sources, their potential benefits will not be fully achieved if there are no suitable means to support interlinking and exchanging this kind of information. This challenge can be addressed by adopting the suite of technologies developed in the Semantic Web, such as Linked Data model and SPARQL. When using these technologies, and with respect to an application scenario which requires managing and querying a vast amount of sensor data, the task of selecting a suitable RDF engine that supports spatio-temporal RDF data is crucial. In this paper, we present our empirical studies of applying an RDF store for Linked Sensor Data. We propose an evaluation methodology and metrics that allow us to assess the readiness of an RDF store. An extensive performance comparison of the system-level aspects for a number of well-known RDF engines is also given. The results obtained can help to identify the gaps and shortcomings of current RDF stores and related technologies for managing sensor data which may be useful to others in their future implementation efforts.

Keywords: IoT, Semantic Web, Linked Data, Sensor, Triple store

1. Introduction

The *Internet of Things*(IoT) is a network of physical objects embedded with sensors that are providing real-time observations about the world as it happens. With

[*]Corresponding author.

an estimation of there being 50 billion connected IoT objects by 2020[16], there will be an enormous amount of sensor observation data being continuously generated per second. Connecting these sensor observation data sources to the rest of the digital world, and turning this data into meaningful actionable information, will create new capabilities, richer experiences and unprecedented economic opportunity for individuals, organisations and states. However, deriving trends, patterns, outliers and unanticipated relationships in such an enormous amount of dynamic data with unprecedented speed and adaptability is extremely challenging.

When trying to fully exploit the huge potential of these sensor data sources, deriving insights from dynamic raw observation data poses various challenges in terms of data integration. Fortunately, a suite of technologies developed through the Semantic Web [6] effort, such as the RDF model, Linked Data [8], SPARQL [36], and RDF stores, can be used as some of the principal solutions to help relieve sensor data sources from the challenge of poor integration [12, 30]. Among these technologies, the RDF store is one that was developed that allows management of RDF data. Additionally, an RDF store typically also provides a public SPARQL endpoint such that data can be queried from RDF-enabled applications via the SPARQL query language.

In the Linked Sensor Data context, the usage of an RDF store is relatively recent. This is due to several specific requirements which are required in order to enable querying and storage of sensor data. In fact, sensor data is always associated with some spatio-temporal contexts, i.e, they are produced in specific locations at specific points in time. Therefore, all sensor data items can be represented in three dimensions: semantic, spatial and temporal. As a result, to enable efficient (and accurate) querying and storage of this multi-dimensional sensor data, spatio-temporal computation support becomes a mandatory requirement for an RDF store or database. Moreover, the "big data" nature of sensor data also requires that these systems need to scale to millions of sensor sources and years of data.

In this paper, the requirements and constraints to adopt an RDF store as a fundamental back-end solution for semantic sensor-based applications and services are analyzed. Moreover, an extensive performance comparison of the most used and well-known RDF store engines that can be utilised for Linked Sensor Data are presented. A requirements analysis, along with detailed evaluation results, can be used to assess the readiness of current RDF database technologies for managing Linked Sensor Data. In summary, the main contributions of this paper are:

1. A detailed analysis on the fundamental requirements for an RDF processing engine that can be applied for Linked Sensor Data.
2. An extensive performance assessment of five selected and well-known RDF stores for managing Linked Sensor Data. In comparison with existing works, our performance study also focuses on evaluating the spatial, temporal and text data indexing performance of these systems.
3. A set of important findings about the gaps and shortcomings of current RDF stores in supporting spatio-temporal computation and full-text searches.
4. Identification of a query optimization challenge for executing a complex spatio-temporal query over Linked Sensor Data.

The remainder of this paper is organized as follows. Section 2 reviews the related work as regards the study and evaluation of existing RDF stores. In Section 3, we analyze the fundamental requirements of RDF stores for Linked Sensor Data. Section 4 describes the general architectural design of current RDF databases that support spatio-temporal querying. A selected set of five popular RDF stores for our analysis is introduced in Section 5. Section 6 presents the general evaluation methodology for assessing the RDF stores (RDF engines) for Linked Sensor Data. Section 7 describes the experimental setting. Section 8 reports on the evaluation results when the evaluation methodology described in Section 6 is carried out. A discussion and our main findings are also given in this section. Finally, we conclude our work in the last section.

2. Related work

During the last decade, we have witnessed a rapid increase in the number of RDF store implementations that have been proposed [2, 7, 15, 20, 33, 37, 41]. Along with this growth is a corresponding increase in studies interested in looking into their performance and data processing behaviours. For that reason, a number of performance assessment techniques specific to these RDF stores have been introduced. For example, in [31], the authors provide a performance comparison of seven selected RDF stores over the synthetic Lehigh University Benchmark (LUBM) dataset [19].

This evaluation aims to test the data loading and query performance of these stores with respect to a large data application. Similarly, Rohloff et al. [38] present a performance evaluation over the LUBM dataset of AllegroGraph, Jena and Sesame with various storage backends (such as MySQL, DAML DB[34], BigIOWLIM, etc). In this work, a set of evaluation metrics is presented such as the data loading time, query execution performance, query completeness and soundness, and storage size requirements. Unlike [31, 38] that compares the performance of different RDF stores, the evaluation described in [39] focuses on the experimental comparison of a single native triple store (Sesame) and a vertically partitioned scheme for storing RDF data in a relational database on top of the SP2Bench SPARQL benchmark [40]. In [9], the authors introduce the Berlin SPARQL Benchmark (BSBM) for comparing the performance of native RDF stores, non-RDF relational databases and SPARQL-to-SQL rewriters. The benchmark provides valuable insights into system behaviour by comparing the data loading time, the number of mixed queries executed per hour, etc.

Regarding an performance study of RDF stores over spatial RDF data, Kolas [26] presents a state-of-the-art assessment for querying geospatial data encoded in RDF. In this work, the author extends the LUBM dataset by adding spatial entities so that they were able to evaluate a spatial search on the geo-enabled RDF stores. Along the same lines, Garbis et al. [18] developed a benchmark, called Geographica, which uses both real-world and synthetic datasets to test the offered functionality and the performance of some prominent geospatial RDF stores. Despite the wide range of performance assessments between RDF stores, to the best of our knowledge, there is no existing approach that focuses on studying the readiness of RDF stores as regards Linked Sensor Data. In comparison with traditional RDF data, sensor data is usually associated with spatial and temporal contexts. This distinctive characteristic therefore poses challenges for the current RDF store implementations due to the requirements of geospatial supports, temporal filtering and full-text search. In this regard, rather than providing another benchmarking effort, our paper should be read as an empirical study to find the gaps and shortcomings with current RDF store technologies so that they can be properly applied for the management of Linked Sensor Data. Following this direction, our evaluation and analysis aims to help guide the development of RDF stores, rather than serving as a comparison of existing ones.

3. Fundamental requirements of a processing engine for linked sensor data

Sensor data has distinctive characteristics that make traditional RDF engines an obsolete solution. This is due to the limited capability of such engines to process the massive amount of linked sensor data available, as well as the lack of spatio-temporal index support. In general, when providing an RDF processing engine for linked sensor data, there are some important features that should be provided. These features are as follows:

- **Geospatial search**: This mandatory feature plays an important role for processing sensor data. Having spatial search support will help to not only provide the spatial information of a spatial object, but will also help to compute the spatial relationships between the two geometries of objects (i.e, within, intersecting, etc). For example, this can be used for finding all the weather stations that are operating within a given area.
- **Temporal filter**: Most of the queries from end-user or sensor-based applications require filtering on the temporal aspect of sensor observation data. This task is very expensive due to the high frequency updates of sensor observation data. Furthermore, this is also the main reason behind a dramatic increase in the required storage size. Therefore, several enhancements should be made to the temporal filter by the query processing engine so that observation data can be easily retrieved.
- **Full-text search**: This is an advanced feature that aims to improve performance of full-text search queries. Essentially, these kind of searches are mostly based on finding given keywords embedded in literal values such as descriptions, street names, location names, etc.
- **Scalability**: This is also another advanced feature that allows the engine to deal with the "big data" processing challenge of sensor data. Also, this can help the engine to address a performance issue when executing a high volume of user queries. Thus, either a scalable solution or an efficient index mechanism are needed [5]. One possible solution that can be considered is one that will duplicate the same service so as to allow many concurrent queries while also providing suitable fault tolerance capabilities. Alternatively, another solution is to split the data across different servers and to provide a middleware

component that can coordinate data transactions amongst these underlying servers.

– **Spatio-temporal query language**: It is important to note that the standard SPARQL query language does not support spatio-temporal queries (nor full-text queries). For this reason, to enable this feature, the RDF database creators must provide spatio-temporal querying by either defining a new language in their own specific syntax or by extending the SPARQL query language.

4. Architectural Design

The general architectural design of RDF stores that support spatio-temporal query processing over Linked Sensor Data can be classified into two categories, namely *native* architectures and *hybrid* architectures. In this section, we will discuss the details for each architecture and analyze the features that might affect the query performance of each.

4.1. Native architectural design

The native architectural design of RDF stores for Linked Sensor Data is illustrated in Figure 1. This architecture needs to implement the data indexing system, the query engine, and physical storage.

4.1.1. Data index

The data index has a vital role and can significantly affect the performance of data insertion. This is one of the primary design elements for storage systems. Additionally, an efficient data indexing system also helps to improve query performance. In the Linked Sensor Data context, along with the triple index, defining spatio-temporal data indices is an essential requirement for engines that follow native architectural design principles. This multidimensional index feature not only helps with triple-based retrieval requirements but also with supporting spatio-temporal queries. For example, Virtuoso uses RTrees for spatial indexing and bitmap indices for RDF triples. Meanwhile, Jena uses a Lucene spatial index along with B/B+ and three advanced triple indexes on *spo*, *pos* and *osp* to accommodate different triple patterns.

Another important feature of the data index component is to define the triple patterns that are used to extract the spatial, temporal or text values from the input data. The extracted values will be indexed properly based on their characteristics and implied context. For example, spatial data are indexed by an R/R+ trees algorithm while the text literals are analyzed by a string index algorithm.

4.1.2. Query engine

The query engine is used for data retrieval which is generally comprised of three sub-modules, namely a query parser, query optimizer and query executor.

Query parser The query parser is responsible for parsing the input query from a user or an application program to an algebraic expression. Unlike the standard SPARQL query parser, the one for spatio-temporal queries over Linked Sensor Data has to adapt to its associated spatio-temporal query language syntax. As SPARQL does not cover spatio-temporal queries, so the query language that supports spatio-temporal or full-text search can be defined either by the RDF engines in their own syntax or by extending the SPARQL language. For example, Virtuoso and Jena define their own spatial query language by adding spatial built-in functions to SPARQL. These additional functions are assigned along with dedicated prefixes, such as <*bif:*> and <*spatial:*>, for Virtuoso and Jena, respectively. In the meantime, instead of defining a new query syntax, Strabon and Stardog adopt the WGS84 and OGC's GeoSPARQL vocabularies, which have been widely used and have become the standard W3C recommendation.

Query optimizer This module is used to determine the most efficient way to execute a given query by considering all the possible query execution plans. The proper execution plan will be represented as an execution-order tree that consists of algebraic operators. A good execution plan will help to prune all the redundant intermediate results, and thus will improve the query performance by reducing the memory consumption as well as result materialization.

Obviously, each query plan leads to different query performance. Finding the proper execution plan that can balance the resource-consuming cost and the query response time is always an ultimate goal of the query optimizer. In the Linked Sensor Data context, this process becomes more complex and expensive due to the difficulties in building a comprehensive cost model that can reflect all the spatial and temporal aspects of Linked Sensor Data. In addition, the lack of statistical information on spatio-temporal data and access patterns also makes the join order optimization challenging and resource consuming. Currently, most RDF query optimizers can only collect limited statis-

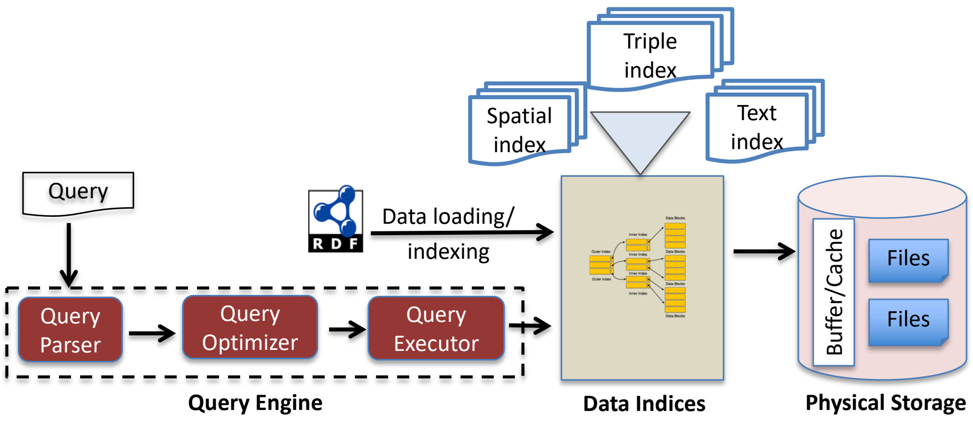

Fig. 1. The native architectural design

tics, such as Jena and Strabon, which collects the number of times a predicate appears. Furthermore, other systems (i.e., Virtuoso and Stardog) cache query plans for later use.

Query executor The query executor is responsible for taking the query execution plan handed back by the query optimizer, recursively processing it by accessing the physical storage, and returning the query results.

4.1.3. *Physical storage*

The final component is the physical storage which includes the database files, a buffer or its own file cache manager that manages the buffering of data to reduce the number of disk accesses. The physical storage can be classified into two types, namely *native* and *non-native* storage [14].

Native storage The *native storage* treats RDF triples as first-class citizens and stores them in a way that is close to the RDF data model. There are two approaches to implement a *native storage*: persistent disk-based and main memory-based. The *persistent disk-based* systems store the RDF data permanently in files [22, 32, 33]. The advantage of these implementations is that the data is safe during a system restart. However, it should be considered that the data writing and reading operations can be slow due to a performance bottleneck.

In contrast to the *disk-based* approaches, the *main memory-based* ones keep data in the system memory. All the data reading and writing operations occur there, thereby improving the overall system performance. However, due to the limited size of the memory, this solution requires a memory-efficient data representation as well as composite index-based techniques so that there will be enough space to not only store the data but also for the other operations associated with query processing and data management [17]. Some implementations falling into this category are Jena, Hexastore[43], Bitmap[3], etc.

Non-native storage The *non-native storage* relies on the relational database system to store RDF data permanently [11, 15, 20, 21, 44]. In this type of storage, data is either stored in a single table (schema-free approach) or in a set of relational tables (schema-based approach) such as a subject table, predicate table, object table, etc. The advantage of the *non-native storage* is the less demanding efforts in terms of design and implementation. Nevertheless, in order to achieve good system performance, an efficient mapping solution of SPARQL-to-SQL and graph-to-relational schema have to be taken into consideration [14].

4.2. *Hybrid architectural design*

The hybrid architecture uses existing systems as sub-components for the processing needed. The common architecture of a hybrid solution is illustrated in Figure 2. In this architecture, the chosen sub-components are accessed with different query languages and different input data formats. Hence, the hybrid approach needs a Query Rewriter, a Query Delegator and a Data Transformer. The Query Rewriter rewrites a SPARQL-like query to sub-queries that the underlying systems can understand. The Query Delegator is used to delegate the execution process by externalizing the processing to sub-systems with the rewritten sub-queries. In some cases, the Query Delegator also includes some components for correlating and aggregating partial results returned from the hybrid databases if they support this. The Data Transformer is responsible for converting input data to the compatible formats of the access methods used in the hybrid databases. It also has to transform the query results from these systems to the format that the Query Delegator requires.

By delegating the processing to available systems, building a system following the hybrid architecture takes less effort than using the native approach. However, the disadvantage of this approach include the limited control over the sub-components as well as the challenges involved to efficiently coordinate them.

5. Analyzing existing RDF stores for Linked Sensor Data

In this section, we will briefly analyze the most popular and mature existing RDF stores that qualify for the required features which have been identified and discussed in Section 3. Table 1 summarizes the features supported by these selected stores. Please be aware that the selection process has been carried out with the target of identifying suitable RDF stores that can be used for efficiently processing Linked Sensor Data with respect to the provided features. A detailed systematic-level evaluation of these stores will be presented later in Section 7.

5.1. Virtuoso Open Source

Virtuoso Open Source (v7.2.4) [15] is an example of the native architecture type and is also a widely-used RDF store in the Semantic Web community. It is the storage system that is hosting the DBPedia database [4]. Virtuoso is a general purpose RDBMS with extensive RDF adaptations that are comprised of RDF-oriented data types and a SPARQL-to-SQL front-end compiler. In Virtuoso, RDF data can be stored as RDF quads which are indexed properly in SQL tables. The engine supports the SPARQL 1.1 language. A SPARQL query will be translated properly to SQL via the SPARQL-to-SQL compiler. Virtuoso's user community is quite active and the software is updated regularly.

The current version of Virtuoso Open Source does not support clustering features. However, it supports advanced spatial indexing (using RTrees) and full-text search. The engine provides its own spatial and text search query language via SPARQL built-in functions. Virtuoso does not have a dedicated temporal index. Instead, the value with a timestamp is indexed as an RDF literal value. Therefore, a temporary query is processed via the provided built-in SPARQL date-time functions. For RDF data, Virtuoso's index scheme consists of five indices (*psog, pogs, sp, op, gs*) that have two full indices over RDF quads plus three partial indices.

Virtuoso serves SPARQL queries via a pre-assigned public SPARQL endpoint. The query modules will then translate an input, SPARQL query to the corresponding SQL query referring to the five triple store tables mentioned above. Virtuoso is backed by a RDBMS and can adopt to all SQL optimization techniques. For example, it provides a cost-based SQL optimizer which performs several types of query transformation, such as join order, index selection, selection of join algorithms, etc. The Virtuoso cost model is based on table row counts, defined indices and uniqueness constraints, and column cardinalities, i.e. counts of distinct values in columns. Additionally, histograms can be made for value distribution of individual columns.

5.2. Stardog

Stardog[1] is a knowledge graph platform that supports RDF storage. Stardog follows the native architecture design solution. Originally, it was implemented by the developers of a well-known OWL reasoner (Pellet) [42]. At the time of writing, the latest version of Stardog is v6.0.1 which supports full-text search, spatial and basic temporal filters. As a graph database, Stardog supports ACID transactions and SPARQL 1.1 [14]. Unlike Virtuoso, this engine does not define its own spatio-temporal query language. Instead, it adopts the WGS84 and OGC's GeoSPARQL vocabularies [35].

There are two indexing modes supported in Stardog, one based only on triples and another one for quads. Indexing data are stored on-disk for both cases. However, "in-memory" mode is also available. No other details in terms of index mechanisms are provided in any publications.

Stardog follows the bottom-up approach to evaluate the query execution plans generated from a given SPARQL query [25]. In the query plan tree, leaf nodes without input are evaluated first, and their results are then sent to their parent nodes up the plan. Typical examples of leaf nodes include scans, i.e. evaluations

[1]http://stardog.com

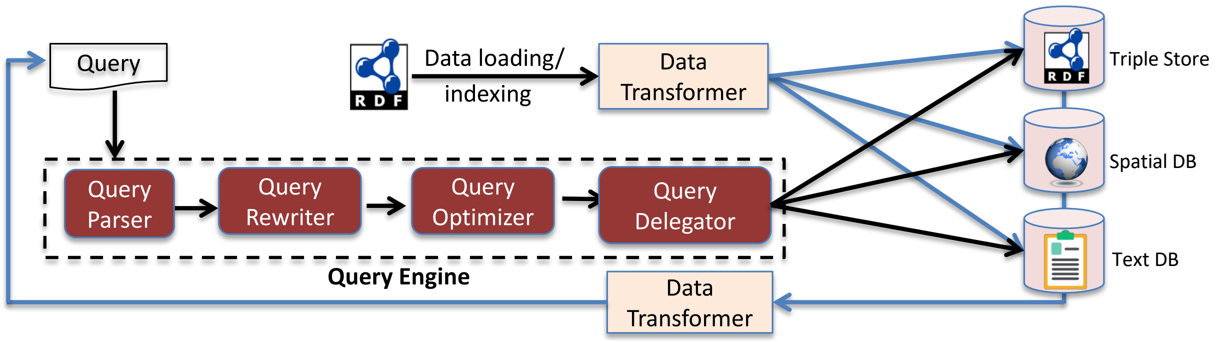

Fig. 2. The hybrid architectural design

of triple patterns, evaluations of full-text search predicates, and VALUES operators. Parent nodes, such as joins, unions, or filters, take the results produced by the leaf nodes as inputs, process them and send their results further towards the root of the tree. The root node of the plan tree, which is typically one of the solution modifiers[2], produces the final results of the query which are then encoded and sent to the client.

Another important feature of Stardog is clustering support which allows horizontal scaling. However, this feature is only available in the commercial version. According to a recent platform report, Stardog can process 10 billion triples stored on a single server.

5.3. Apache Jena

Apache Jena[3] is an open source SPARQL 1.1 framework for Java. It provides an API to extract data from and write to RDF graphs. Jena implements the native architecture design which includes the optional quads RDF storage layers, namely TDB (file system), SDB (SQL DBMS), and in memory. Jena also provides inference support (supporting RDFS, OWL-Lite or using custom rules), but it works only on triple stores and not on quadruples stores. From version 3, Jena supports full-text search and basic spatial index through the use of Lucene or Solr. No clustering solution has been mentioned.

Regarding the indexing architecture, the one in Jena is built around three concepts, namely a node table, triple/quad indexes, and a prefixes table. Among them, the node table is responsible for storing the dictionary which provides two mappings for the RDF terms, namely *string-to-id* and *id-to-string*. The default storage of the former is implemented using B+ trees, and the latter is based on a sequential access file. Triples and quads are stored in specialized structures. Triples are held as thre identifiers in the node table while quads are assigned as four. Again, B+ trees are used to persist these indices.

Query execution in Jena involves both static and dynamic optimizations [1]. Static optimizations aim to transform the algebras of the execution tree into new, equivalent and optimized algebra forms. This transformation process is performed in advance of the query execution. Meanwhile, dynamic optimizations involve deciding on the best execution approach during the execution phase and can take into account the actual data so far retrieved.

A number of optimization strategies are provided: a statistics-based strategy, a fixed strategy and a strategy of no reordering. For the statistics-based strategy, the Jena optimizer uses information captured in a per-database statistics file to specify the join order of the query triple patterns. For that, the statistics file takes the form of a number of rules for approximate matching counts for triple patterns. The file can be automatically generated when the engine starts. Users can update it manually by adding and modifying rules to tune the database based on higher-level knowledge, such as inverse function properties.

5.4. RDF4J

RDF4J[4] (formerly known as Sesame [10]) is an open source Java framework for processing RDF data. It is a native RDF processing engine which includes

[2]https://www.w3.org/TR/sparql11-query/#solutionModifiers
[3]https://jena.apache.org/
[4]http://rdf4j.org

parsing, storing, inferencing and querying over RDF. Similar to Apache Jena, RDF4J also supports two out-of-the-box RDF databases (in-memory and native store) along with third party storage solutions. It also fully supports SPARQL 1.1, full-text search and spatial index in conjunction with the GeoSPARQL language. RDF4J has no clustering feature.

The key component of RDF4J is a “Storage And Inference Layer” (SAIL). In short, SAIL is an application programming interface that offers RDF-specific methods to its client and translates these methods to calls to its specific underlying DBMS. The benefit of the introduction of SAIL is the flexible implementation of RDF4J on top of a wide variety of repositories without changing other RDF4J’s components. Consequently, in addition to its own *in-memory* and *disk-based* repositories, RDF4J can be easily attached to other DBMS such as MySQL, PostgreSQL, etc. The repository used in this paper is the default native *disk-based* repository, which originally provided by RDF4J. Regarding the query optimization used in RDF4J, no detailed information are publicly provided.

5.5. Strabon

Strabon[28] is an open-source semantic DBMS that can be used to store linked geospatial data expressed in stRDF format and query it using the stSPARQL language [27]. It follows the hybrid architecture design and is backed by a PostGIS extension of Postgres RDBMS plus RDF4J, allowing it to manage both semantic and spatial RDF data. Strabon supports SPARQL 1.1. It provides spatial, temporal search and basic text search. In contrast to other engines, Strabon provides an advanced temporal search which allows users to query Allen’s temporal relationship between two temporal objects. In terms of indexing capability, the developers of Strabon reported that it can scale up to 500 million triples. No clustering solution is available for Strabon.

When data is imported into to Strabon, the data is firstly decomposed into stRDF triples and each triple is processed separately. Data is stored using dictionary encoding that follows a “per-predicate” scheme. The “per-predicate” scheme uses vertical partitioning to store triples in different tables based on their predicate, and one table per predicate is maintained. The B-Tree algorithm is then applied on these predicate tables. For spatial literals found during the data loading, Strabon stores them in a dedicated *geo_values* table. A spatial index is then created by applying an R-tree-over-GiST algorithm [23].

Query processing in Strabon is implemented by modifying the RDF4J components. The query engine consists of a parser, an optimizer, an evaluator and a transaction manager. Moreover, besides the standard formats offered by Sesame, Strabon offers the KML and GeoJSON encodings, which are widely used for representing the spatial data. Strabon applies exactly the same as Sesame’s query optimization techniques for the standard SPARQL part of a stSPARQL query. For the spatial extension functions, an additional optimization step is introduced. In this step, the cost of the spatial functions presented in the query will be evaluated by PostGIS. Based on the cost estimated, the query tree is then optimized and executed.

6. Evaluation methodology and metrics

The previous sections give insight to the architecture and available features of selected RDF stores that can be applied for Linked Sensor Data. This section will describe in detail the methodology and the metrics used to evaluate the performance of these systems. We have triggered the performance of Jena, RDF4J and Strabon by modifying their source code. Virtuoso and Stardog already provide some metrics themselves, which we retrieve by using the Virtuoso JDBC/CLI and Stardog APIs, respectively.

Our evaluation methodology is carried out within two phases. In the first phase, we will evaluate the data loading performance of the RDF stores over the linked sensor dataset. In this experiment, we evaluated separately the loading performance of semantic, spatial and text data.

The second phase is to evaluate the query execution performance. We firstly define a benchmark queries set which is performed over our linked meteorological dataset. These queries and dataset are described later in Section 7. After having these benchmark queries defined, we evaluate the query performance by executing these queries. In addition to the overall query execution time, we also measure the execution performance of query parsing, query optimization and query execution. The query parsing time is calculated from the time the system retrieves the query string to the time the query algebra tree is generated. Similarly, the query optimization process is considered starting from the time the query tree and it is finished when an execution plan is delivered. For simplicity, any other run-time de-

Table 1
RDF stores comparison

	Technical characteristics						
	Query language	**Spatial filter**	**Full-text search**	**Temporal filter**	**License**	**Clustering**	**Latest release Date**
Virtuoso 7	SPARQL, SQL	Y	Y	Y (basic)	Cm,OS	N	Oct 2018
Stardog 6	SPARQL, SQL	Y	Y	Y (basic)	Cm,OS	Y	Jan 2019
Apache Jena	SPARQL	Y	Y	Y (basic)	OS	N	March 2019
RDF4J	SPARQL	Y	Y	Y (basic)	OS	N	April 2019
Strabon	SPARQL	Y	N	Y (advance)	OS	N	Jan 2018

cisions are considered as part of the query execution rather than part of the query optimization. The query execution is finished when the last result has been received.

7. Experimental Settings

7.1. Benchmark dataset

Our experiments are conducted over the linked meteorological dataset, a sub-set of our GoT dataset [29]. The meteorological dataset consists of over 30 years of meteorological data for over the world. The dataset is categorized into two parts, namely static and dynamic dataset. The static one describes the station and sensor descriptions, such as spatial information, text data, etc, and is not frequently updated. The dynamic dataset contains the observation data generated by the sensor station described in the static dataset. The dynamic dataset is frequently updated.

Due to the large size of the meteorological dataset and also because of our limited infrastructure resources, only a subset of this dataset is used in our experiments. As described in Table 2, this subset contains the static dataset and the observation data from year 2016 to 2018, results in a set of 2.5B triples.

The spatial and text data loading performance are evaluated over the static dataset. To provide a more comprehensive evaluation, we enlarge this dataset by generating more spatial and text objects on the basis of the SOSA/SSN data model [13, 24], same data model used for our linked meteorological data. The static dataset is described in Table 3.

The dynamic dataset presenting the observation data is used to evaluate the query performance in the second phase of our experiment. The observation data are extracted within the time period from Jan 2016 to May 2016 (250M triples), as described in Table 4, which we believe could indicate a general performance measure for this test.

Table 2
Benchmark Linked Meteorological Dataset

Year	**Num. Observation**	**Num. Triples (billion)**	**Size (Gb)**
2016	79,305,738	0.555	112
2017	194,748,696	1.363	274
2018	357,237,456	2.5	503
Sum	**631,291,890**	**4.419**	**889**

Table 3
Static datasets description

Dataset	Static dataset
Num. spatial/text object (million)	20
Num. triples (million)	120
Raw Data size	20G

7.2. Benchmark queries

We have selected a set of 11 queries that performs over our benchmark dataset. In general, our benchmark queries aim to test the engine processing capability with respect to their provided features for querying Linked Sensor Data. As previously mentioned, because the standard SPARQL 1.1 language does not support spatio-temporal queries nor the full-text queries, some RDF stores have to extend the SPARQL language with their own specific syntax. Therefore, some of these queries need to be rewritten so that they are compatible with the engine under test.

We summarized some highlight features of the benchmark queries as follows: (i) if the query has input parameter; (ii) if it requires geo-spatial search; (iii) if it uses temporal filter; (iv) if it uses full-text search on string literal; (v) if it has Group By feature; (vi) if the results needs to be ordered via ORDER By operator; (vii) if the results are using the LIMIT operator; (viii) the number of variables in the query; and (ix) the number of triples patterns in the query. The group-by, order-by and limit operators imply the effectiveness of the query optimization techniques used by the engine

Table 4
Datasets description for query performance evaluation (from 01/2016 to 05/2016)

Dataset	**Num. observation (million)**	**Num. Triples (million)**	**Size**
2 months	17.5	122.4	2.8G
3 months	22.2	155.6	4.3G
4 months	25.8	180.9	6.5G
5months	30.5	213.3	7.1G
Static	A	0.75	135Mb

(e.g., parallel unions, ordering or grouping using indexes, etc.), and the number variables and triples patterns give a measure of query complexity. These summary of highlight features are described in Table 5 and their SPARQL representations are presented in the Appendix 9.

7.3. Platform

We conducted our experiments on a physical server which has following configuration: 2x E5-2609 V2 Intel Quad-Core Xeon 2.5GHz 10MB Cache, Hard Drive 3x 2TB Enterprise Class SAS2 6Gb/s 7200RPM - 3.5" on RAID 0, Memory 128GB 1600MHz DDR3 ECC Reg w/Parity DIMM Dual Rank.

7.4. Setup

We have experimented on Jena v3.9.0, RDF4J v2.6.5, Stardog v6.0.1 and Virtuoso Open Source v7.2.5. For Virtuoso, the system parameters NumberOfBuffers and MaxDirtyBuffers were set to 40GB. Java heap size for Jena and RDF4J is also set to 40GB. Besides, for Jena, we have enabled its optimization strategy option to statistic strategy. Regarding RDF4J configuration, we have configured the index mechanism to *spoc*, *posc* and *opsc*. For Stardog, as we will not evaluate the inference capability, we disable this option when importing data. The spatial and textual index options were enabled. Similar to other engines, Stardog memory size was also set to 40 GB. The rest of the parameters were left to the default values.

8. Results and Discussion

In this section we present and discuss the experimental results following the evaluation methodology and metrics described in Section 6.

8.1. Data loading throughput

8.1.1. Triple loading performance

The triple loading performance of the test stores is depicted in Figure 3. We can see how the performance is affected when the size of the RDF dataset increases. According to the results, Virtuoso is the fastest, followed by Stardog and Jena with about 2 times and 6 times slower than Virutoso, respectively . RDF4J is the slowest being about 10 times slower than Virtuoso. According to our observation, there are two possible explanations for the poor loading performance of RDF4J: (1) The high frequent index updates due to the relatively small page size used by RDF4J; (2) The HTTP communication latency between the RDF4J client and server components. Note that, RDF4J does not support bulk data import. Instead, it provides a set of REST API functions for client to access the RDF database either for reading, writing or querying data. This consequently leads to the dramatical increase in the network communication latency time when the number of request access to the database goes up. This reason is also explained for the poor data loading performance of Strabon.

Regarding the required repository size, it can be observed in Table 6 that this metric has a linear increase with respect to the dataset size. Unfortunately, due to the complex hybrid architecture of Strabon, its repository size metric can not be measured. For Virtuoso, its index compression methods pay off, resulting in the smallest index size. Stardog uses 80% more space than Virtuoso. Jena and RDF4J generate much larger indexes about 4 times larger than those of Virtuoso, though they have less indexes.

8.1.2. Spatial and text data loading performance

Unlike the existing performance studies that only focus on general data loading performance, our evaluation also measures the loading performance on spatial and text data. In this regard, we measure the loading speed via the number of object can be indexed per second, instead of the number of triples. An object con-

Table 5
Benchmark queries characteristics

Query	Parametric	Spatial filter	Temporal filter	Text search	Group By	Order By	LIMIT	Num. variables	Num. triple patterns
1	✓	✓						3	3
2	✓							3	4
3	✓		✓					7	8
4	✓	✓	✓		✓	✓	✓	7	8
5	✓		✓		✓	✓		4	5
6	✓		✓					5	8
7	✓	✓	✓			✓	✓	7	8
8				✓		✓		3	4
9				✓	✓			6	7
10	✓		✓	✓	✓			7	9
11					✓			2	1

Table 6
Required repository capacity (in GB) with respect to dataset size

RDF Engine/Data size	500 (mil)	750 (mil)	1000 (mil)	1250 (mil)	1500 (mil)	1750 (mil)	2000 (mil)
RDF4j	60	79	99	121	142	NA	NA
Virtuoso 7	14	19	23	32	39	47	56
Stardog	24	43	64	82	102	125	146
Apache Jena	62	89	117	135	164	NA	NA
Strabon	NA	NA	NA	NA	NA	NA	NA

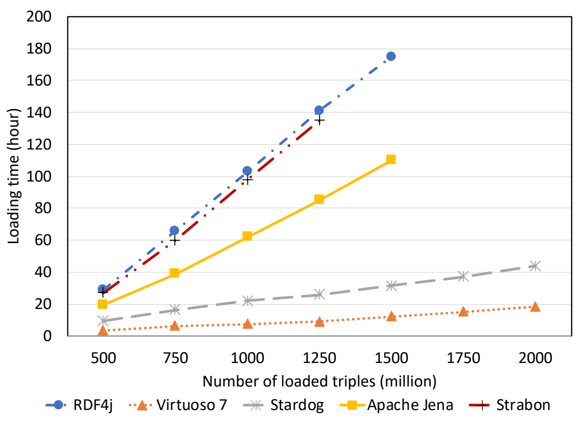

Fig. 3. RDF stores performance of RDF data loading

sists of spatial and text description. This evaluation helps us to have a better understanding about the indexing behaviour of the test engines for specific types of data such as geo spatial and text.

The spatial and text data loading time with respect to dataset size is shown in Figure 4. Figure 5 depicts the average loading speed. As described, RDF4J and Strabon perform poorly - their average loading speed is less then 1000 object/sec when loading 20 millions objects. Apache Jena has a better performance and its loading speed can reach to 1306 object/sec. Its loads 20 (mil) dataset in about 5.6 hours. Stardog performs better than Jena, results in 4 hours for loading the same dataset and the average speed is almost 14k object/sec. In comparison, Virtuoso achieves much faster loading speed than others, which can reach to 3287 object/sec. However, a drawback of the impressive loading speed is the increase of number triples stored in Virtuoso with respect to the original data provided. This is because the additional *geo:geometry* spatial triples which are generated during the transformation of the *geo:lat* and *geo:long* triples to enable the geo-spatial indexing.

8.2. *Query performance*

The set of figures 6 to 16 presents the average query response time concerning Jena, Virtuoso, RDF4J, Stardog and Strabon with respect to the different time horizons of two, three, four and five months observation data of year 2016, respectively. The datasets used to evaluate the query performance are described in Table 4. The benchmark queries have been executed by performing a pseudo-random sequence of these queries repeated ten times in order to minimize the caching effect. Beside, we also empty the system cache before running the query of each test execution.

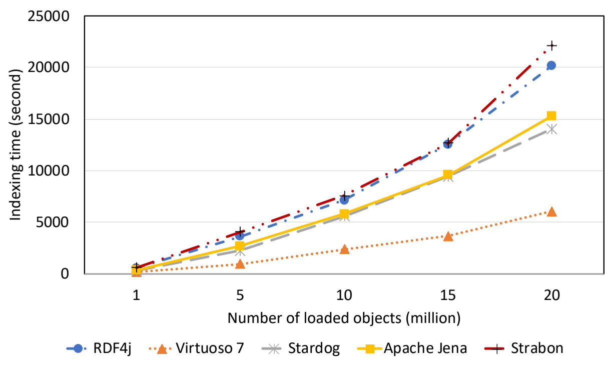

Fig. 4. Spatial/text data indexing time

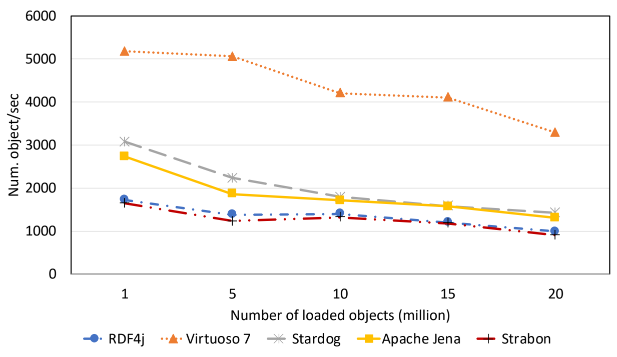

Fig. 5. Average spatial/text data indexing speed

Looking at the evaluation results, unsurprisingly, the query performance of all systems decreases with the growth of dataset. However, in the cases of no spatio-temporal nor full-text searches are involved (Q2, Q11), the query performances remain stable and are weakly effected by the increase of dataset size. This is understandable because these queries are only performed over the static dataset, which is unchanged and not influenced by the dynamic data. Virtuoso and Stardog perform closely and require less execution time, in the order of *ms*. For example, in the case of 5 months dataset, Virtuoso takes 83 (ms) to execute the Q2 while it is 137 (ms) in Stardog. Regarding the others, the performances of RDF4j and Strabon are comparable, followed by Jena.

For the query that has only spatial filter and basic triple matching such as Q1, Virtuoso executes this query in about 80(ms) for 5 months dataset. Following this is Stardog with 112(ms). Jena and RDF4J appear to have the worst performance with average response times are 50128(ms) and 1761(ms), respectively. We attribute this to the effectiveness of the query optimizer in each system. Wrong execution plan might lead to a catastrophic performance. We will discuss this further in the following subsection that is about the cost breakdown of query performance.

Another aspect to be considered is the mixing of spatial filter, time filter and full-text search queries (Q3, Q4, Q5, Q9, Q10). According to the evaluation results, Virtuoso is still best ranked with less then 1(s) of execution time for 5 months dataset, followed by Stardog. With Jena, RDF4J and Strabon, we obtained an extremely high query execution time, greater than 50(s). Analyzing the cost breakdown of these queries, we derive two possible explanations: (1) The growth of the observation dataset size, which certainly requires more data processing operations (look up, read, etc) and resource-consuming, results in the considerable increase of the query execution time. (2) The inefficient query execution strategies that have been applied. Inappropriate execution plan also consequently effects to the query performance.

8.3. *Cost breakdown*

Table 7 shows the cost breakdown between query parsing, query optimization and execution, across all stores and queries for 5 months dataset. It is easy to realize that, except Virtuoso, for most stores, the query run-time cost is mostly dominated by the execution time. For example, looking at the Q3 cost breakdown, the execution costs of Jena and Stardog are 48,833 (ms) and 11410 (ms), taking over 97% and 78% of query response time, respectively. Meanwhile, Virtuoso dominates only 74 (ms) which takes about 12% the total cost.

The query parsing time only takes a small fraction of the total cost. This is applied for all stores. Regarding the cost of query optimization, we observe that this cost is different across the systems. Specifically, Virtuoso spends significantly more time with respect to others. An example of this is the Q6, Virtuoso spends 48 (ms) over the total time 49 (ms) for the query optimization, taking almost 100%. In contrast, the optimization cost of Q6 is 80 (ms) in Stardog, about 55% of the total cost.

In the following, for further analyzing the cost breakdown, we choose the Q10 as the representative for all queries. The reason for this is due to its high complexity, that covers almost all query characteristics and also demands a heavy computation. Analyzing the results in Figure 17, we observes that: (1) Optimization costs of this query in Virtuoso, Stardog and Jena are fairly consistent and are not significantly affected

Table 7: Cost breakdown of evaluated queries for 5 months dataset

Query	**Jena**			**Virtuoso**			**RDF4j**			**Stardog**			**Strabon**		
	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)	Pars. (ms)	Opt. (ms)	Exe. (ms)
Q1	10	1814	48304	1	75	4	9	542	1210	2	101	9	11	551	88
Q2	7	105	53	1	70	12	6	1100	127	1	94	42	8	100	1203
Q3	15	1320	48833	1	500	74	16	2500	226414	1	3085	11410	20	3140	59830
Q4	13	1840	48305	2	560	27	15	1432	309528	1	3115	10883	19	4859	55427
Q5	14	1410	48739	1	50	4	15	90	14	2	73	66	21	85	6
Q6	13	163	10	1	48	0	16	813	356	1	80	64	23	90	815
Q7	20	1731	48415	2	72	4	21	1430	227404	1	204	3	30	2147	56817
Q8	10	159	373	1	65	4	8	1100	868	1	71	41	15	1425	568
Q9	16	1774	48388	1	110	8	12	2113	151653	1	105	136	17	3549	76977
Q10	22	1677	48470	2	635	8	21	1546	144141	2	3093	11201	33	1187	71636
Q11	5	37	410	1	15	0	4	20	4	1	46	67	6	19	3

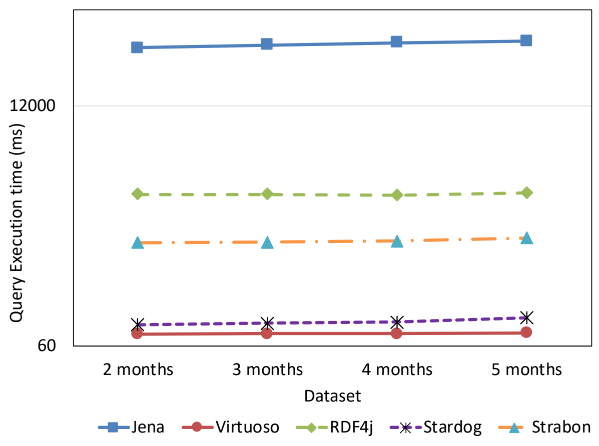

Fig. 6. Q1 execution time

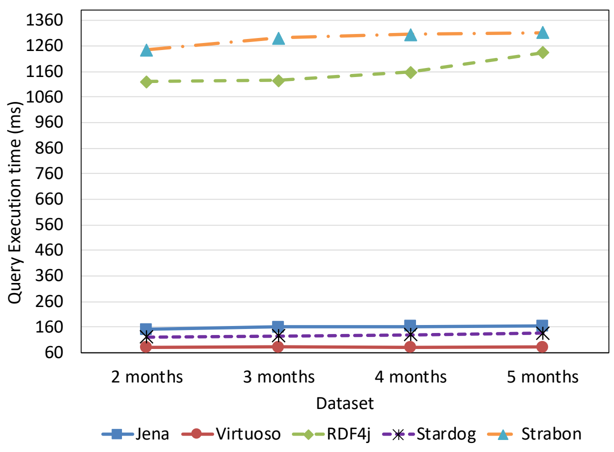

Fig. 7. Q2 execution time

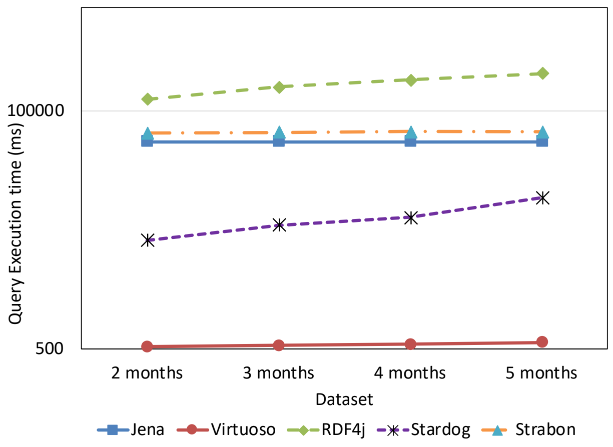

Fig. 8. Q3 execution time

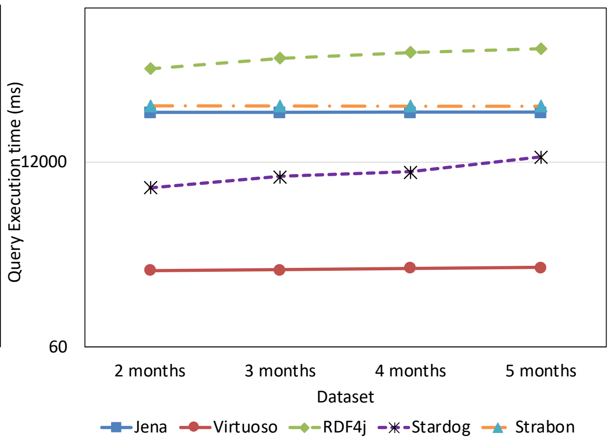

Fig. 9. Q4 execution time

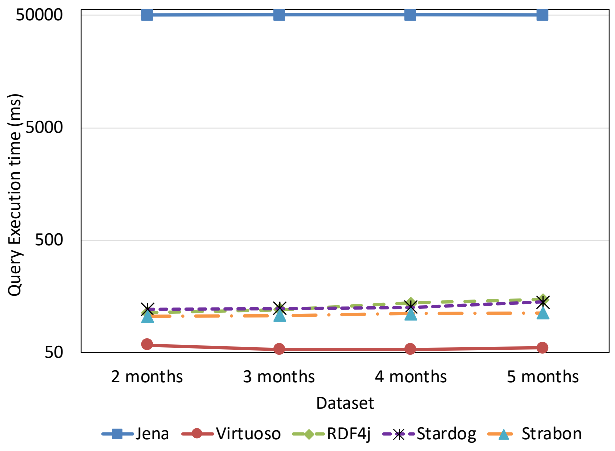

Fig. 10. Q5 execution time

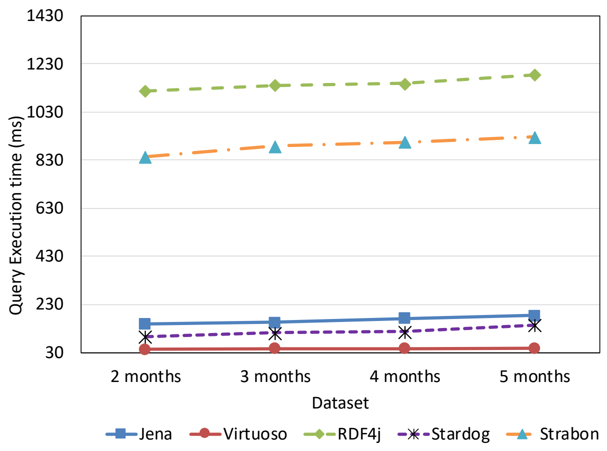

Fig. 11. Q6 execution time

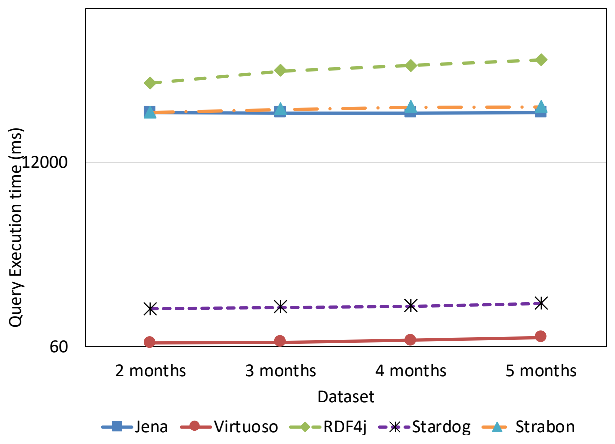

Fig. 12. Q7 execution time

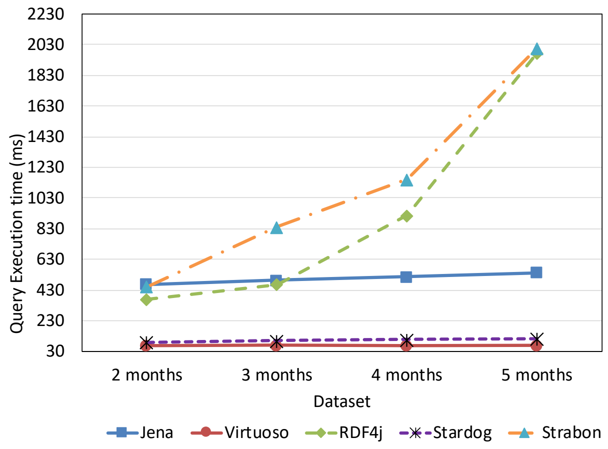

Fig. 13. Q8 execution time

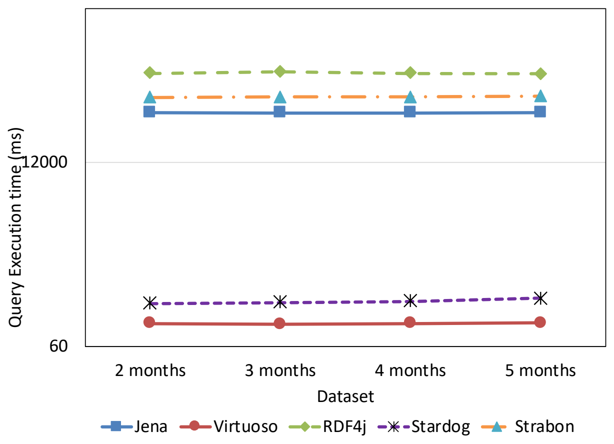

Fig. 14. Q9 execution time

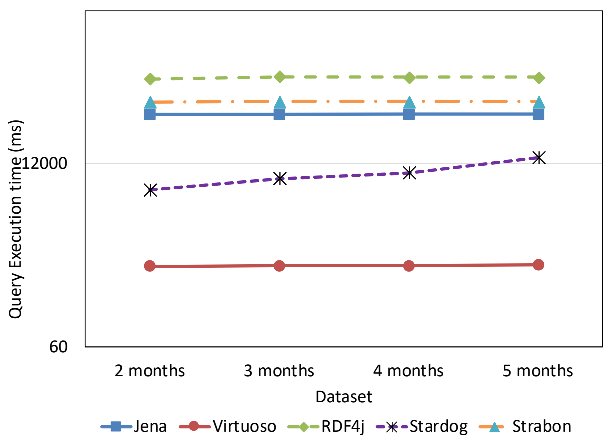

Fig. 15. Q10 execution time

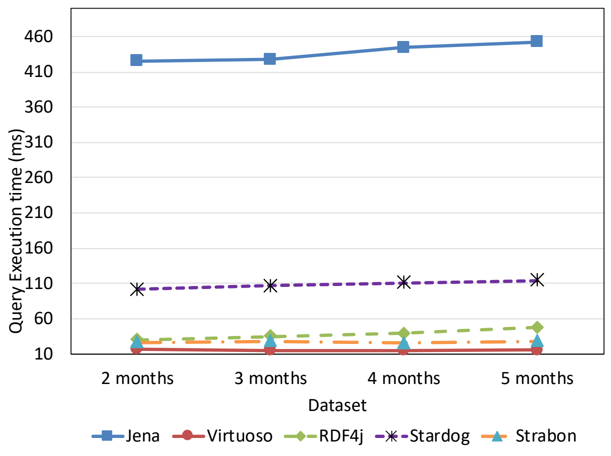

Fig. 16. Q11 execution time

by the dataset size. We attribute this to the effectiveness of query execution plan caching and the statistical approach taken in these systems. (2) On the contrary, RDF4J and Strabon clearly present a growth in optimization costs with respect to the dataset size. This indicates that these two systems have applied different optimization techniques in the presence of different workloads.

Following these two observations mentioned above, it still lacks of evidences to conclude which strategy performs better. It is evidenced by the unpredictable query processing behaviour that draws our attention on Q1. Note that, the complexity of this query is low as it only requires simple spatial computation in conjunction with semantic triple matching. Among the four stores, Virtuoso spends 98% query run-time for query optimization. For Q1, the total query run-time is 80(ms) and Virtuoso takes 75(ms) for query optimization. However, the significant effort for optimization pays off. Figure 18 illustrates Q1 execution costs across different stores, where Virtuoso significantly outperforms other stores. In the meanwhile, RDF4J spends only 30% time for query optimization on Q1, much less than Virtuoso, but the execution time is extremely high, which is 1210(ms), far worse than Virtuoso.

In the same case of Q1, Apache Jena spends more time than RDF4J for query optimization, 1814(ms) in comparison with 542(ms). However, it results in a catastrophic performance. Analyzing the Jena query processing log, we attribute this to the inappropriate query execution plan generated for Q1. According to the data statistic, a most efficient way to execute Q1 is that the spatial filter with the low selectivity should be executed first. Thereafter the results will be joined with the semantic triple matching with higher selectivity. This execution plan helps to eliminate all unnecessary intermediate results so that it will reduce the number of join operations. Unfortunately, Jena executes this query in the opposite way, as shown in Listing 1, that requires a lot of join operations. This results in the dramatical increase of query processing time. We attribute this to a lack of spatio-temporal statistical information about the dataset. Although Jena already provided a function to generate the statistic of a specified dataset, however, it only works for standard RDF dataset, not for spatio-temporal dataset.

```
(join
  (quadpattern
   (quad <urn:x-arq:DefaultGraphNode>
    ?station rdf:type got:ontology/WeatherStation)
     (quad <urn:x-arq:DefaultGraphNode>
        ?station geo:hasGeometry ?geoFeature)
   (propfunc spatial:withinCircle
     ?geoFeature (59.783 5.35 20 "miles")
     (table unit)
))
```

Listing 1: Jena optimization plan for Q1

8.4. Discussion

Our performance study addresses a number of well-know RDF stores such as Virtuoso, RDF4J, Apache Jena, Stardog and Strabon in terms of data loading performance and query execution behaviours on processing over Linked Sensor Data. In order to analyze the systems strengths and weakness of these stores, we first carefully assess their data loading performance on sensor data. In this regards, along with the general RDF data loading evaluation, we also study the spatial and text data loading speed. Generally, Virtuoso and Stardog perform better than others. Moreover, we also learn that they have better supports on bulk and near real-time data import. In our opinion, this is an advanced feature that should be considered when selecting a RDF engine for sensor data.

Along with the different data aspects loading assessment, corresponding query performance evaluations are also discussed. Unlike the existing approaches that only observe the overall query performance, we also analyze the dynamics and query processing behaviours of the test stores at a deeper detailed level. We split the query execution process into a series of sub processes, which are query parsing, query optimization and execution. Each sub process is then evaluated separately. Through the evaluation results, with the general assessment, Virtuoso outperforms all other stores. It is followed by Stardog and Strabon. Moreover, we also learn that the time cost for each process is different across these stores. For example, Virtuoso dedicates significantly most time for query optimization process with respect to the others. On the contrary, Jena and RDF4J mostly focus on the execution process.

Beside the judgments about the overall engines performance, several further findings related to data index and query optimization have been derived: (1) Due to the poor performance of the queries that requires analytical computing on sensor observation data, such

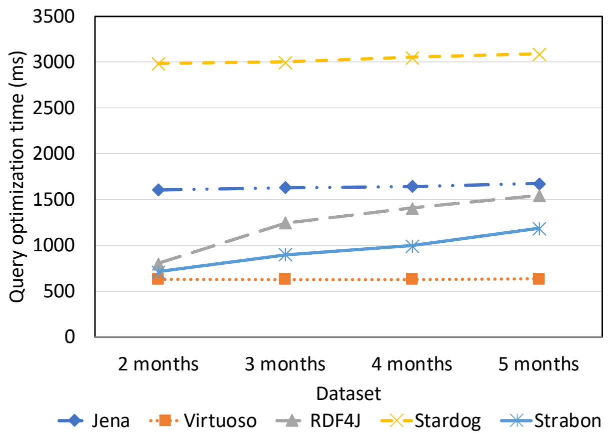

Fig. 17. The optimization time of Query 10 by varying increasing dataset

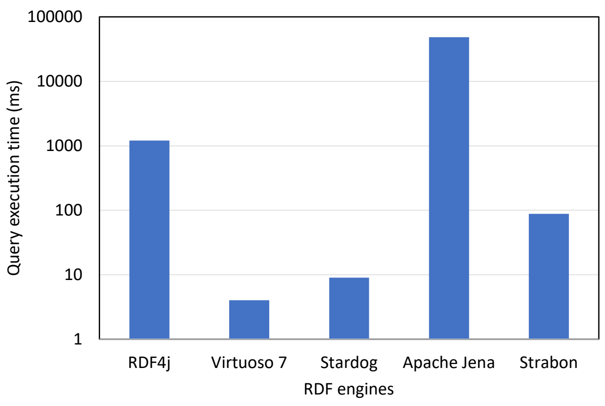

Fig. 18. Q1 execution costs for 5 months dataset (in logscale)

as aggregation, a proper time series index approach is needed. Strabon has a dedicated temporal index for data with time-stamp, but it just focuses on querying temporal relations of two temporal objects rather than on supporting efficient analytical functions . (2) Selecting a wrong query execution plan is potentially catastrophic. In our experiments, for Jena, failure in query planning results in the worse performance. (3) There is a lack of statistic data on spatial and temporal aspects of sensor dataset. This is evidenced by the inefficient query planning (Q1,Q3, Q4, Q5) of the RDF stores that use statistic optimization strategy such as Jena and RDF4J.

9. Conclusion and Future Work

This paper presents our recent efforts to provide a comprehensive performance study of RDF stores for Linked Sensor Data. One of the goals of our paper is to summarize the list of fundamental requirements of RDF stores so that they can be used for managing and querying sensor data. We have also described the abstract architecture design of current RDF database technologies that support spatio-temporal queries. Another main contribution of our work is to provide a comparative analysis of data loading and query performance of a representative set of RDF stores, namely Apache Jena, Virtuoso, RDF4J, Stardog and Strabon. In our evaluation, particular attentions have been given on evaluating the performance of geo-spatial search, temporal filter and full-text search over sensor data. Since such assessment aspects have not been fully considered and addressed by the existing works, our paper gives valuable insights about the strengths and weaknesses of the currents RDF stores implementation when applying for Linke Sensor Data.

For future work, we are currently planning more experiments on spatio-temporal distributed RDF stores. We want to have a more comprehensive understanding about the spatio-temporal query processing behaviours in a distributed environment. Our long-term goal is to develop of a scalable spatio-temporal query processing engine for Linked Sensor Data.

Acknowledgements

This paper has been funded in part by Science Foundation Ireland under grant numbers SFI/12/RC/2289 and SFI/16/RC/3918 (co-funded by the European Regional Development Fund), the BIG-IoT project under grant number 688038, and the Marie Skłodowska-Curie Programme SMARTER project under grant number 661180.

Appendix - Benchmark Queries SPARQL Representations

PREFIXES

```
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX got:<http://graphofthings.org/ontology/>
PREFIX geoname: <http://www.geonames.org/ontology#>
```

Query 1. Given the latitude and longitude position, it retrieves the nearest weather station within 10 miles

```
SELECT ?station ?coor
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    FILTER (<bif:st_intersects>
        (?coor, <bif:st_point>($long$, $lat$),$radius$)).
}
```

Query 2. Given the country name, it retrieves the total number of weather station deployed in this country

```
SELECT (count(?station) as ?total)
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    ?geoFeature geoname:parentCountry "$country␣name$".
}
```

Query 3. Given the country name and year, it detects the minimum temperature value that has been observed for that specified year

```
SELECT min(?value) as ?min  ?station
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentCountry "$country␣name$".
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:SurfaceTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
                  sosa:resultTime ?time;
                    sosa:hasSimpleResult ?value.
    filter (year(?time)=$year$).
}
```

Query 4. Given the area location and radius, it detects the hottest month of that area in given year

```
SELECT ?month (avg(?value) as ?avgTemp)
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    FILTER (<bif:st_intersects>(?coor, <bif:st_point>
                            ($long$,$lat$),$radius$)).
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:SurfaceTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
            sosa:resultTime ?time;
            sosa:hasSimpleResult ?value.
    filter (year(?time)=$year$).
}
GROUP BY (month(?time) as ?month)
ORDER BY DESC (avg(?value)) limit 1
```

Query 5. Given the station URI and year, it retrieves the average wind speed for each month of year

```
SELECT ?month (avg(?value) as ?avgTemp)
WHERE
{
    ?sensor sosa:isHostedBy <$station␣URI$>.
    ?sensor sosa:observes got:WindSpeedProperty.
    ?obs sosa:madebySensor  ?sensor;
         sosa:resultTime ?time;
         sosa:hasSimpleResult ?value.
    filter (year(?time)=$year$)
}
GROUP BY (month(?time) as ?month)
ORDER BY ?month
```

Query 6. Given a date, it retrieves the total number of observation that were observed in California state

```
SELECT count(?obs) as ?number
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentADM1 "California".
    ?geoFeature geoname:parentCountry "United␣States".
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:SurfaceTemperatureProperty.
    ?obs sosa:madebySensor  ?sensor;
                sosa:resultTime ?time.
    filter (year(?time)=$year$ && month(?time)=$month$
                && day(?time)=$day$).
}
```

Query 7. Given the latitude, longitude and radius , it retrieves the latest visibility observation value of that area

```
SELECT ?value ?time
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature wgs84:geometry ?coor.
    FILTER (<bif:st_intersects>(?coor, <bif:st_point>
                                ($long$,$lat$),$radius$)).
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:AtmosphericVisibilityProperty.
    ?obs sosa:madebySensor  ?sensor;
          sosa:resultTime ?time;
          sosa:hasSimpleResult ?value.
}
ORDER BY DESC (?time)
LIMIT 1
```

Query 8. Given a keyword, it retrieves all the places matching a keyword

```
SELECT ?station ?place ?sc
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentADM1 ?place.
    ?place bif:contains "'$keyword$'" OPTION (score ?sc).
}ORDER BY ?sc
```

Query 9. Given a place name prefix, it summaries the number of observation of places that match a given keyword. The results are grouped by place and observed property

```
SELECT count(?obs) as ?totalNumber ?place ?observedType
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentCountry ?place.
    ?place bif:contains "'$name_prefix$'*" OPTION (score ?sc).
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes ?observedType.
    ?obs sosa:madebySensor  ?sensor.
}GROUP BY ?place ?observedType
```

Query 10. Given a keyword, it retrieves the average humidity value for places that matches a keywords since 2016

```
SELECT avg(?value) as ?avgValue ?place
WHERE
{
    ?station a got:WeatherStation.
    ?station geo:hasGeometry ?geoFeature.
    ?geoFeature geoname:parentCountry ?place.
    ?place bif:contains "'$keyword$*'" OPTION (score ?sc).
    ?sensor sosa:isHostedBy ?station.
    ?sensor sosa:observes got:AtmosphericPressureProperty.
    ?obs sosa:madebySensor  ?sensor;
            sosa:resultTime ?time;
            sosa:hasSimpleResult ?value.
    filter(year(?time)>=2016)
}GROUP BY ?place
```

Query 11. It retrieves the total number of sensor for each observed properties

```
SELECT (count(?sensor) as ?number) ?obsType
WHERE
{
    ?sensor sosa:observes ?obsType
}GROUP BY ?obsType
```

References

[1] Tdb optimizer. https://jena.apache.org/documentation/tdb/optimizer.html.

[2] Jans Aasman. Allegro graph: Rdf triple database. *Cidade: Oakland Franz Incorporated*, 17, 2006.

[3] Medha Atre, Jagannathan Srinivasan, and James A Hendler. Bitmat: A main memory rdf triple store. *Tetherless World Constellation, Rensselar Plytehcnic Institute, Troy NY*, 2009.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[5] Pierfrancesco Bellini and Paolo Nesi. Performance assessment of rdf graph databases for smart city services. *Journal of Visual Languages & Computing*, 45:24–38, 2018.

[6] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 2001.

[7] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. Owlim: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[8] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (ldow2008). In *Proceedings of the 17th World Wide Web*. ACM, 2008.

[9] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[10] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

[11] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment, 2005.

[12] Michael Compton, Cory Andrew Henson, Laurent Lefort, Holger Neuhaus, and Amit P Sheth. A survey of the semantic specification of sensors. 2009.

[13] Michael Compton et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.

[14] Olivier Curé and Guillaume Blin. *RDF database systems: triples storage and SPARQL query processing*. Morgan Kaufmann, 2014.

[15] Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[16] D. Evans. The internet of things: How the next evolution of the internet is changing everything. 2011.

[17] David C Faye, Olivier Cure, and Guillaume Blin. A survey of rdf storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 15:11–35, 2012.

[18] George Garbis, Kostis Kyzirakos, and Manolis Koubarakis. Geographica: A benchmark for geospatial rdf stores (long version). In *International Semantic Web Conference*, pages 343–359. Springer, 2013.

[19] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[20] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. 2003.

[21] Stephen Harris and Nigel Shadbolt. Sparql query processing with conventional relational database systems. In *International Conference on Web Information Systems Engineering*, pages 235–244. Springer, 2005.

[22] Andreas Harth and Stefan Decker. Yet another rdf store: Perfect index structures for storing semantic web data with contexts. Technical report, DERI Technical Report, 2004.

[23] Joseph M Hellerstein, Jeffrey F Naughton, and Avi Pfeffer. *Generalized search trees for database systems*. September, 1995.

[24] Krzysztof Janowicz, Armin Haller, Simon JD Cox, Danh Le Phuoc, and Maxime Lefrançois. Sosa: A lightweight ontol-

ogy for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56:1–10, 2019.

[25] Pavel Klinov. How to read stardog query plans. https://www.stardog.com/blog/how-to-read-stardog-query-plans/.

[26] Dave Kolas. A benchmark for spatial semantic web systems. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2008.

[27] M. Koubarakis and K. Kyzirakos. Modeling and querying metadata in the semantic sensor web: the model strdf and the query language stsparql. In *Proc. ESWC*, volume 12, pages 425–439, 2010.

[28] Kostis Kyzirakos et al. Strabon: a semantic geospatial dbms. In *ISWC*. Springer, 2012.

[29] Danh Le-Phuoc et al. The graph of things: A step towards the live knowledge graph of connected things. *Journal of Web Semantics*, 37, 2016.

[30] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Josiane Xavier Parreira, and Manfred Hauswirth. The linked sensor middleware–connecting the real world and the semantic web. *Proceedings of the Semantic Web Challenge*, 152:22–23, 2011.

[31] Baolin Liu and Bo Hu. An evaluation of rdf storage systems for large data applications. In *2005 First International Conference on Semantics, Knowledge and Grid*, pages 59–59. IEEE, 2005.

[32] Akiyoshi Matono, Said Mirza Pahlevi, and Isao Kojima. Rdfcube: A p2p-based three-dimensional index for structural joins on distributed triple stores. In *Databases, Information Systems, and Peer-to-Peer Computing*, pages 323–330. Springer, 2006.

[33] Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[34] Zhengxiang Pan and Jeff Heflin. Dldb: Extending relational databases to support semantic web queries. Technical report, LEHIGH UNIV BETHLEHEM PA DEPT OF COMPUTER SCIENCE AND ELECTRICAL ENGINEERING, 2004.

[35] Matthew Perry and John Herring. Ogc geosparql-a geographic query language for rdf data. *OGC implementation standard*, 2012.

[36] Eric Prud, Andy Seaborne, et al. Sparql query language for rdf. 2006.

[37] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, page 4. ACM, 2012.

[38] Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder, and John Sumner. An evaluation of triple-store technologies for large data stores. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1105–1114. Springer, 2007.

[39] Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, and Christoph Pinkel. An experimental comparison of rdf data management approaches in a sparql benchmark scenario. In *International Semantic Web Conference*, pages 82–97. Springer, 2008.

[40] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Spˆ 2bench: a sparql performance benchmark. In *2009 IEEE 25th International Conference on Data Engineering*, pages 222–233. IEEE, 2009.

[41] Michael Sintek and Malte Kiesel. Rdfbroker: A signature-based high-performance rdf store. In *European Semantic Web Conference*, pages 363–377. Springer, 2006.

[42] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[43] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[44] Kevin Wilkinson and Kevin Wilkinson. Jena property table implementation, 2006.