dipLODocus$_{[RDF]}$—Short and Long-Tail RDF Analytics for Massive Webs of Data

Marcin Wylot, Jigé Pont, Mariusz Wisniewski, and Philippe Cudré-Mauroux

eXascale Infolab
University of Fribourg, Switzerland
{firstname.lastname}@unifr.ch

Abstract. The proliferation of semantic data on the Web requires RDF database systems to constantly improve their scalability and transactional efficiency. At the same time, users are increasingly interested in investigating or visualizing large collections of online data by performing complex analytic queries. This paper introduces a novel database system for RDF data management called dipLODocus$_{[RDF]}$, which supports both transactional and analytical queries efficiently. dipLODocus$_{[RDF]}$ takes advantage of a new hybrid storage model for RDF data based on recurring graph patterns. In this paper, we describe the general architecture of our system and compare its performance to state-of-the-art solutions for both transactional and analytic workloads.

1 Introduction

Despite many recent efforts, the lack of efficient infrastructures to manage RDF data is often cited as one of the key problems hindering the development of the Semantic Web. Last year at ISWC, for instance, the two industrial keynote speakers (from the New York Times and Facebook) pointed out that the lack of an open-source, efficient and scalable alternative to MySql for RDF data was the number one problem of the Semantic Web.

The Semantic Web community is not the only one suffering from a lack of efficient data infrastructures. Researchers and practitioners in many other fields, from business intelligence to life sciences or astronomy, are currently crumbling under gigantic piles of data they cannot manage or process. The current crisis in data management is from our perspective the result of three main factors: i) rapid advances in CPU and sensing technologies resulting in very cheap and efficient processes to create data ii) relatively slow advances in primary, secondary and tertiary storage (PCM memories and SSD disks are still expensive, while modern SATA disks are singularly slow–with seek times between 5ms and 10ms typically) and iii) the emergence of new data models and new query types (e.g., graph reachability queries, analytic queries) that cannot be handled properly by legacy systems. This situation resulted in a variety of novel approaches to solve specific problem, for large-scale batch-processing [10], data warehousing [20], or array processing [8].

L. Aroyo et al. (Eds.): ISWC 2011, Part I, LNCS 7031, pp. 778–793, 2011.

Nonetheless, we believe that the data infrastructure problem is particularly acute for the Semantic Web, because of its peculiar and complex data model (which can be modeled as a constrained graph, as a ternary or n-ary relation, or as an object-oriented model depending on the context) and of the very different types of queries a typical SPARQL end-point must support (from relatively simple transactional queries to elaborate business intelligence queries). The recent emergence of distributed Linked Open Data processing and visualization applications relying on complex analytic and aggregate queries is aggravating the problem even further.

In this paper, we propose dipLODocus$_{[RDF]}$, a new system for RDF data processing supporting both simple transactional queries and complex analytics efficiently. dipLODocus$_{[RDF]}$ is based on a novel hybrid storage model considering RDF data both from a graph perspective (by storing RDF subgraphs or RDF molecules) and from a "vertical" analytics perspective (by storing compact lists of literal values for a given attribute). dipLODocus$_{[RDF]}$ trades insert complexity for analytics efficiency: isolated inserts and simple look-up are relatively complex in our system due to our hybrid model, which on the other hand enables us to considerably speed-up complex queries.

The rest of this paper is structured as follows: we start by discussing related work in Section 2. Section 3 gives a high-level overview of our system and introduces our hybrid storage scheme. We give a more detailed description of the various data structures in dipLODocus$_{[RDF]}$ in Section 4. We describe how our system handles common operation like bulk inserts, updates, and various types of queries in Section 5. Section 6 is devoted to a performance evaluation study, where we compare the performance of dipLODocus$_{[RDF]}$ to state-of-the-art systems both for a popular Semantic Web benchmark and for various analytic queries. Finally, we conclude in Section 7.

2 Related Work

Approaches for storing RDF data can be broadly categorized in three subcategories: triple-table approaches, property-table approaches, and graph-based approaches. Many approaches have been proposed to optimize RDF query processing; we list below some of the most popular approaches and systems. We refer the reader to recent surveys of the field (such as [15], [13], or [16]) for a more comprehensive coverage.

Triple-Table Storage: since RDF data can be seen as sets of *subject-predicate-object* triples, many early approaches used a giant triple table to store all data. Our GridVine [2,7] system, for instance, uses a triple-table storage approach to distribute RDF data over decentralized P2P networks using the P-Grid [1] distributed hash-table. More recently, Hexastore [21] suggests to index RDF data using six possible indices, one for each permutation of the set of columns in the triple table, leading to shorter response times but also a worst-case five-fold increase in index space. Similarly, RDF-3X [17] creates various indices from a giant triple-table, including indices based on the six possible permutations of the

triple columns, and aggregate indices storing only two out of the three columns. All indices are heavily compressed using dictionary encoding and byte-wise compression mechanisms. The query executor of RDF-3X implements a dedicated cost-model to optimize join orderings and determine the cheapest query plan automatically.

Property-Table Storage: various approaches propose to speed-up RDF query processing by considering structures clustering RDF data based on their *properties*. Wilkinson *et al.* [22] propose the use of two types of property tables: one containing clusters of values for properties that are often co-accessed together, and one exploiting the type property of subjects to cluster similar sets of subjects together in the same table. Chong *et al.* [6] also suggest the use of property tables as materialized views, complementing a primary storage using a triple-table. Going one step further, Abadi *et al.* suggest a fully-decomposed storage model for RDF: the triples are in that case rewritten into n two-column tables where n is the number of unique properties in the data. In each of these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects. The authors then advocate the use of a column-store to compactly store data and efficiently resolve queries.

Graph-Based Storage: a number of further approaches propose to store RDF data by taking advantage of its graph structure. Yan *et al.* [23] suggest to divide the RDF graph into subgraphs and to build secondary indices (e.g., Bloom filters) to quickly detect whether some information can be found inside an RDF subgraph or not. BitMat [4] is an RDF data processing system storing the RDF graph as a compressed bit matrix structure in main-memory. gStore [24] is a recent system storing RDF data as a large, labeled, and directed multi-edge graph; SPARQL queries are then executed by being transformed into subgraph matching queries, that are efficiently matched to the graph using a novel indexing mechanism.
Several of the academic approaches listed above have also been fully implemented, open-sourced, and used in a number of projects (e.g., GridVine[1], Jena[2], and RDF-3X[3]).

A number of more industry-oriented efforts have also been proposed to store and manage RDF data. Virtuoso[4] is an object-relational database system offering bitmap indices to optimize the storage and processing of RDF data. Sesame[5] [5] is an extensible architecture supporting various back-ends (such as PostgreSQL) to store RDF data using an object-relational schema. Garlik's 4Store[6] is a parallel RDF database distributing triples using a round-robin approach. It stores triple in triple-tables (or quadruple-tables more precisely). BigOWLIM[7] is a scalable RDF database taking advantage of ordered indices and data statistics to optimize

[1] http://lsirwww.epfl.ch/GridVine/
[2] http://jena.sourceforge.net/
[3] http://www.mpi-inf.mpg.de/neumann/rdf3x/
[4] http://virtuoso.openlinksw.com/
[5] http://www.openrdf.org/
[6] http://4store.org/
[7] http://www.ontotext.com/owlim/

queries. AllegroGraph[8], finally, is a native RDF database engine based on a quadruple storage.

3 System Rationale

Our own storage system in dipLODocus$_{[RDF]}$ can be seen as a hybrid structure extending several of the ideas from above. Our system is built on three main structures: RDF molecule clusters (which can be seen as hybrid structures borrowing both from property tables and RDF subgraphs), template lists (storing literals in compact lists as in a column-oriented database system) and an efficient hash-table indexing URIs and literals based on the clusters they belong to.

Figure 1 gives a simple example of a few molecule clusters—storing information about students—and of a template list—compactly storing lists of student IDs. Molecules can be seen as *horizontal* structures storing information about a given object instance in the database (like rows in relational systems). Template lists, on the other hand, store *vertical* lists of values corresponding to one *type* of object (like columns in a relational system). Hence, we say that dipLODocus$_{[RDF]}$ is a *hybrid* system, following the terminology used for approaches such as Fractured Mirrors [19] or our own recent Hyrise system [12].

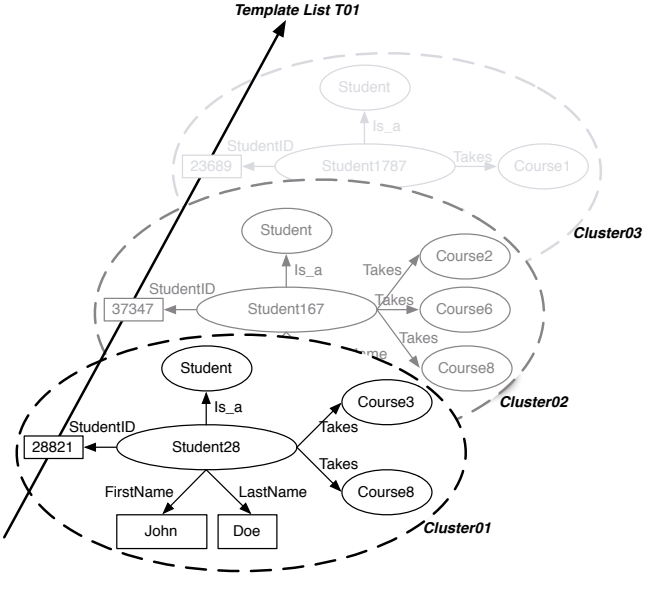


Fig. 1. The two main data structures in dipLODocus$_{[RDF]}$: molecule clusters, storing in this case RDF subgraphs about students, and a template list, storing a list of literal values corresponding to student IDs

[8] http://www.franz.com/agraph/allegrograph/

Molecule clusters are used in two ways in our system: to logically group sets of related URIs and literals in the hash-table (thus, pre-computing joins), and to physically co-locate information relating to a given object on disk and in main-memory to reduce disk and CPU cache latencies. Template lists are mainly used for analytics and aggregate queries, as they allow to process long lists of literals efficiently. We give more detail about both structures below as we introduce the overall architecture of our system.

4 Architecture

Figure 2 gives a simplified architecture of dipLODocus$_{[RDF]}$. The *Query Processor* receives the query from the client, parses it, optimizes it, and creates a query plan to execute it. The *hash-table* uses a lexicographical tree to assign a unique numeric key to each URI, stores metadata associated to that key, and points to two further data structures: the molecule *clusters*, which are managed by the *Cluster Manager* and store RDF sub-graphs, and the *template lists*, managed by the *Template Manager*. All data structures are stored on disk and are retrieved using a page manager and buffered operations to amortize disk seeks. Those components are described in greater detail below.

4.1 Query Processor

The query processor receives inserts, updates, deletes and queries from the clients. It offers a SPARQL [18] interface and supports the most common features

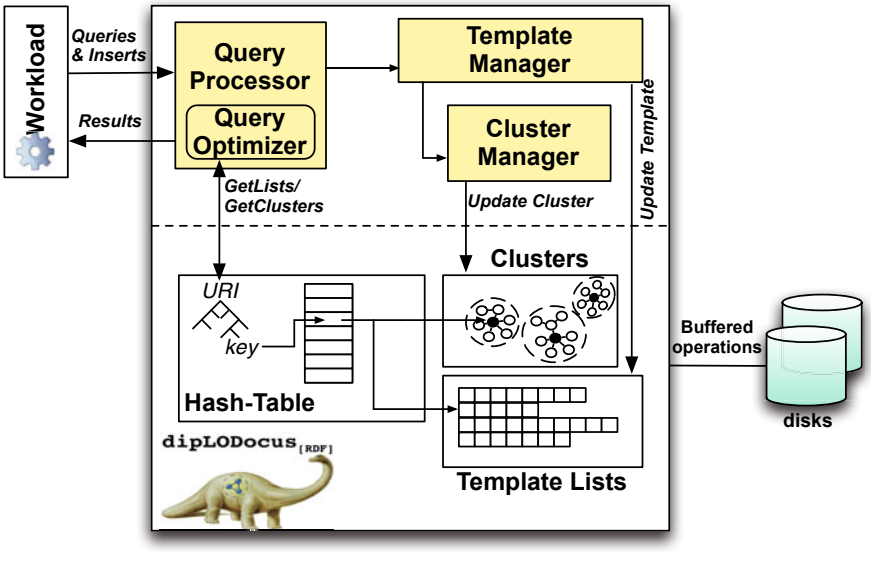


Fig. 2. The architecture of dipLODocus$_{[RDF]}$

of the SPARQL query language, including conjunctions and disjunctions of triple patterns and aggregate operations. We use the RASQAL RDF Query Library[9] to parse both incoming triples serialized in XML, as well as to parse SPARQL queries. New triples are then handed to the Template and Cluster managers to be inserted into the database. As for incoming queries, after being parsed, they are rewritten as query trees in order to be executed. The query trees are passed to the *Query Optimizer*, which rewrites the queries to optimize their execution plans (cf. below Section 5). Finally, the queries are resolved bottom-up, by executing the leaf-operators first in the query tree. Examples of query processing are given below in Section 5.

4.2 Template Manager

One of the key innovations of dipLODocus$_{[RDF]}$ revolves around the use of *declarative storage patterns* [9] to efficiently co-locate large collections of related values on disk and in main-memory. When setting-up a new database, the database administrator may give dipLODocus$_{[RDF]}$ a few hints as to how to store the data on disk: the administrator can give a list of triple patterns to specify the *root nodes*, both for the template lists and the molecule clusters (see for instance above Figure 1, where "Student" is the root node of the molecule, and "StudentID" is the root node for the template list). Cluster roots are used to determine which clusters to create: a new cluster is created for each instance of a root node in the database. The clusters contain all triples departing from the root node when traversing the graph, until another instance of a root node is crossed (thus, one can join clusters based on the root nodes). Template roots are used to determine which literals to store in template lists.

In case the administrator gives no hint about the root nodes, the system inspects the templates created by the template manager (see below) and takes all classes as molecule roots and all literals as template roots (this is for example the case for the performance evaluation we describe in Section 6). Optimizing the automated selection of root nodes based on samples of the input data and an approximate query workload is a typical automated design problem [3] and is the subject of future work.

Based on the storage patterns, the template manager handles two main operations in our system: i) it maintains a schema of triple templates in main-memory and ii) it manages template lists. Whenever a new triples enters the system, it is passed to the template manager, which associates template IDs corresponding to the triple by considering the type of the subject, the predicate, and the type of the object. Each distinct list of "(subject-type, predicate, object-type)" defines a new triple template. The triple templates play the role of an instance-based RDF schema in our system. We don't rely on the explicit RDF schema to define the templates, since a large proportions of constraints (e.g., domains, ranges) are often omitted in the schema (as it is for example the case for the data we consider in our experiments, see Section 6). In case a new template is detected (e.g., a new predicate is used), then the template manager updates its

[9] `http://librdf.org/rasqal/`

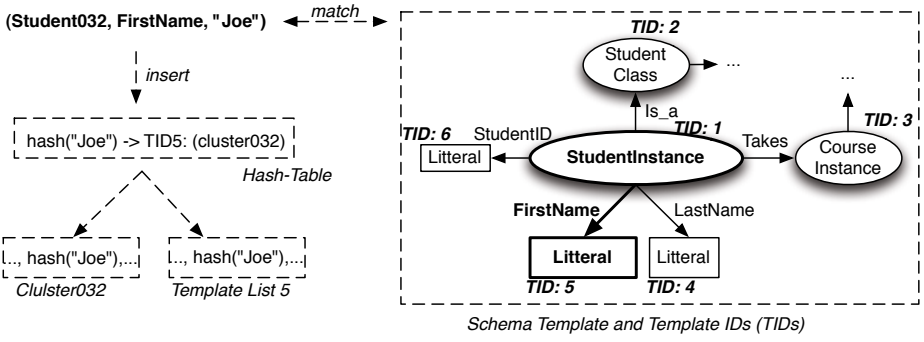


Fig. 3. An insert using templates: an incoming triple (upper left) is matched to the current RDF template of the database (right), and inserted into the hash-table, a cluster, and a template list

in-memory triple template schema and inserts new template IDs to reflect the new pattern it discovered. Figure 3 gives an example of a template. In case of very inhomogeneous data sets containing millions of different triple templates, wildcards can be used to regroup similar templates (e.g., "Student - likes - *"). Note that this is very rare in practice, since all the datasets we encountered so far (even those in the LOD cloud) typically consider a few thousands triple templates at most.

The triple is then passed to the Cluster Manager, which inserts it in one or several molecules. If the triple's object corresponds to a root template list, the object is also inserted into the template list corresponding to its template ID. Templates lists store literal values along with the key of their corresponding cluster root. They are stored compactly and segmented in sublists, both on disk and in main-memory. Template lists are typically sorted by considering a lexical order on their literal values—though other orders can be specified by the database administrator when he declares the template roots. In that sense, template lists are reminiscent of *segments* in a column-oriented database system. Finally, the triple is inserted into the hash-table as well (see Figure 3 for an example).

4.3 Cluster Manager

The Cluster Manager takes care of updating and querying the molecule clusters. When receiving a new triple from the Template Manager, the cluster manager inserts it in the corresponding cluster(s) by interrogating the hash-table (see Figure 3). In case the corresponding cluster does not exist yet, the Cluster Manager creates a new molecule cluster, inserts the triple in the molecule, and inserts the cluster in the list of clusters it maintains.

Similarly to the template lists, the molecule clusters are serialized in a very compact form, both on disk and in main-memory. Each cluster is composed of two parts: a list of offsets, containing for each template ID in the molecule the offset

at which the keys corresponding for the template ID are stored, and the list of keys themselves. Thus, the size of a molecule, both on-disk and in main-memory, is $\#TEMPLATES + (KEY\_SIZE * \#TRIPLES)$, where KEY_SIZE is the size of a key (in bytes), $\#TEMPLATES$ is the number of templates IDs in the molecule, and $\#TRIPLES$ is the number of triples in the molecule (we note that this storage structure is much more compact than a standard list of triples). To retrieve a given information in a molecule, the system first determines the position of the template ID corresponding to the information sought in the molecule (e.g., "FirstName" is the sixth template ID for the "Student" molecule above in Figure 3). It then jumps to the offset corresponding to that position (e.g., 6^{th} offset in our example), reads that offset and the offset of the following template ID, and finally retrieves all the keys/values between those two offsets to get all the values corresponding to that template ID in the molecule.

4.4 Hash-Table

The hash-table is the central index in dipLODocus$_{[RDF]}$; the hash-table uses a lexicographical tree to parse each incoming URI or literal and assign it a unique numeric key value. The hash-table then stores, for every key and every template ID, an ordered list of all the clusters IDs containing the key (e.g., "key 10011, corresponding to a Course object [template ID 17], appears in clusters 1011, 1100 and 1101"; see also Figure 3 for another example). This may sound like a pretty peculiar way of indexing values, but we show below that this actually allows us to execute many queries very efficiently simply by reading or intersecting such lists in the hash-table directly.

5 Common Operations

Given the main components and data structures described above, we describe below how common operation such as inserts, updates, and triple pattern queries are handled by our system.

5.1 Bulk Inserts

Inserts are relatively complex and costly in dipLODocus$_{[RDF]}$, but can be executed in a fairly efficient manner when considered in bulk; this is a tradeoff we are willing to make in order to speed-up complex queries using our various data structures (see below), especially in a Semantic Web or LOD context where isolated inserts or updates are from our experience rather infrequent.

Bulk insert is a n-pass algorithm (where n is the deepest level of a molecule) in dipLODocus$_{[RDF]}$, since we need to construct the RDF molecules in the clusters (i.e., we need to materialize triple joins to form the clusters). In a first pass, we identify all root nodes and their corresponding template IDs, and create all clusters. The subsequent passes are used to join triples to the root nodes (hence, the student clusters depicted above in Figure 1 are built in two phases, one for the Student root node, and one for the triples directly connected to the Student).

During this operation, we also update the template lists and the hash-table incrementally. Bulk inserts have been highly optimized in dipLODocus$_{[RDF]}$, and use an efficient page-manager to execute inserts for large datasets that cannot be kept in main-memory.

5.2 Updates

As for other hybrid or analytic systems, updates can be relatively expensive in dipLODocus$_{[RDF]}$. We distinguish between two kinds of updates: in-place and complex updates. In-place updates are punctual updates on literal values; they can be processed directly in our system by updating the hash-table, the corresponding cluster, and the template lists if necessary. Complex updates are updates modifying object properties in the molecules. They are more complex to handle than in-place updates, since they might require a rewrite of a list of clusters in the hash-table, and a rewrite of a list of keys in the molecule clusters. To allow for efficient operations, complex updates are treated like updates in a column-store (see [20]): the corresponding structures are flagged in the hash-table, and new structures are maintained in write-optimized structures in main-memory. Periodically, the write-optimized structures are merged with the hash-table and the clusters on disk.

5.3 Queries

Query processing in dipLODocus$_{[RDF]}$ is very different from previous approaches to execute queries on RDF data, because of the three peculiar data structures in our system: a hash-table associating URIs and literals to template IDs and cluster lists, clusters storing RDF molecule clusters in a very compact fashion, and template lists storing compact lists of literals. We describe below how a few common queries are handled in dipLODocus$_{[RDF]}$.

Triple Patterns: Triple patterns are relatively simple in dipLODocus$_{[RDF]}$: they are usually resolved by looking for a bound-variable (URI) in the hash-table, retrieving the corresponding cluster numbers, and finally retrieving values from the clusters when necessary. Conjunctions and disjunctions of triples patterns can be resolved very efficiently in our system. Since the RDF nodes are logically grouped by clusters in the hash-table, it is typically sufficient to read the corresponding list of clusters in the hash table (e.g., for "return all students following Course0"), or to intersect or take the union of several lists of clusters in the hash table (e.g., for "return all students following Course0 whose last names are Ḋoé") to answer the queries. In most cases, no join operation is needed since joins are implicitly materialized in the hash-table and in the clusters. When more complex join occurs, dipLODocus$_{[RDF]}$ resolves them using standard hash-join operators.

Molecule Queries: Molecule queries or queries retrieving many values/ instances around a given instance (for example for visualization purposes) are

also extremely efficient in our system. In most cases, the hash-table is invoked to find the corresponding cluster, which contains then all the corresponding values. For bigger scopes (such as the ones we consider in our experimental evaluation below), our system can efficiently join clusters based on the various root nodes they contain.

Aggregates and Analytics: Finally, aggregate and analytic queries can also be very efficiently resolved by our system. Many analytic queries can be solved by first intersecting lists of clusters in the hash-table, and then looking up values in the remaining molecule clusters. Large analytic or aggregate queries on literals (such as our third analytic query below, returning the names of all graduate students) can be extremely efficiently resolved by taking advantage of template lists (containing compact and sorted lists of literal values for a given template ID), or by filtering template lists based on lists of cluster IDs retrieved from the hash-table.

6 Performance Evaluation

To evaluate the performance of our system, we compared it to various RDF database systems. The details of the hardware platform, the data sets and the workloads we used are give below.

6.1 Hardware Platform

All experiments were run on a HP ProLiant DL360 G7 server with two Quad-Core Intel Xeon Processor E5640, 6GB of DDR3 RAM and running Linux Ubuntu 10.10 (Maverick Meerkat). All data were stored on recent 1.4 TB Serial ATA disk.

6.2 Data Sets

The benchmark we used is one of the oldest and most popular benchmarks for Semantic Web data called Lehigh University Benchmark (LUBM) [14]. It provides an ontology describing universities together with a data generator and fourteen queries. We used two data sets, the first one consisting of ten LUBM universities (1'272'814 distinct triples, 315'003 distinct strings), and the second regrouping one hundred universities (13'876'209 distinct triples, 3'301'868 distinct strings).

6.3 Workload

We compared the runtime execution for LUBM queries and for three analytic queries inspired by an RDF analytic benchmark we recently proposed (the BowlognaBench benchmark [11]). LUBM queries are criticized by some for their reasoning coverage; this was not an issue in our case, since we focused on RDF DB query processing rather than on reasoning capabilities. We keep an in-memory representation of subsumption trees in dipLODocus$_{[RDF]}$ and rewrite queries automatically to support subclass inference for the LUBM queries. We manually

rewrote inference queries for the systems that do not support such functionalities (e.g., RDF-3X).

The three additional analytic/aggregate queries that we considered are as follows: 1) a query returning the professor who supervises the most Ph.D. students 2) a query returning a big molecule containing all triples within a scope of 2 of Student0 and 3) a query returning all graduate students.

6.4 Methodology

As for other benchmarks (e.g., tpc-x[10]) we include a warm-up phase before measuring the execution time of the queries. We first run all the queries in sequence once to warm-up the systems, and then repeat the process ten times (i.e., we run in total 11 batches containing all the queries in sequence for each system). We report the mean values for each query and each system below as well as a 95% confidence interval on run times. We assumed that the maximum time for each query shouldn't exceed 2 hours (we stopped the tests if one query took more than two hours to be executed). We compared the output of all queries running on all systems to ensure that all results were correct.

We tried to do a reasonable optimization job for each system, by following the recommendations given in the installation guides for each system. We did not try to optimize the systems any further, however. We performed no fine-tuning or optimization for dipLODocus$_{[\mathrm{RDF}]}$.

We avoided the artifact of connecting to the server, initializing the DB from files and printing results for all systems; we measured instead the query execution times only.

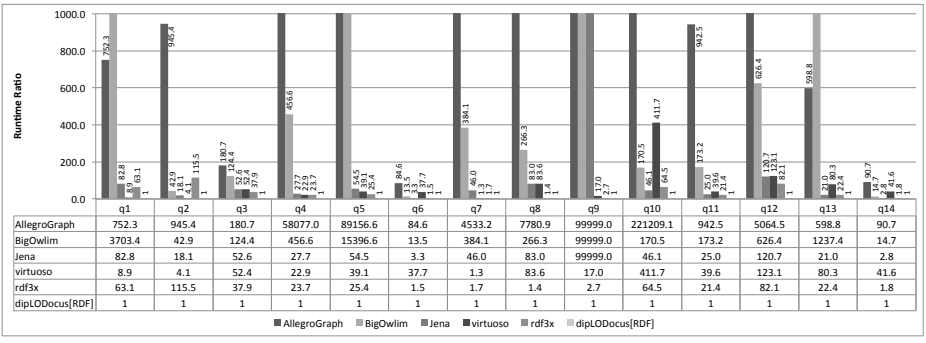
6.5 Systems

We compared our prototype implementation of dipLODocus$_{[\mathrm{RDF}]}$ to five other well-known database systems: Postgres, AllegroGraph, BigOWLIM, Jena, Virtuoso, and RDF 3X. We chose those systems to have different comparison points using well-known systems, and because they were all freely available on the Web. We give a few details about each system below.

Postgres: We used Postgres 8.4 with Redland RDF Library 1.0.13; Postgres is a well-known relational database, but as the numbers below show, it is not optimized for RDF storage. We couldn't run our 100-universities on it because its load time took more than one week. It also had huge difficulties to cope with some of the queries for the 10-universities data set. Since the time of query execution was particularly long for this system, we ran each query five times only and simply report the best run below.

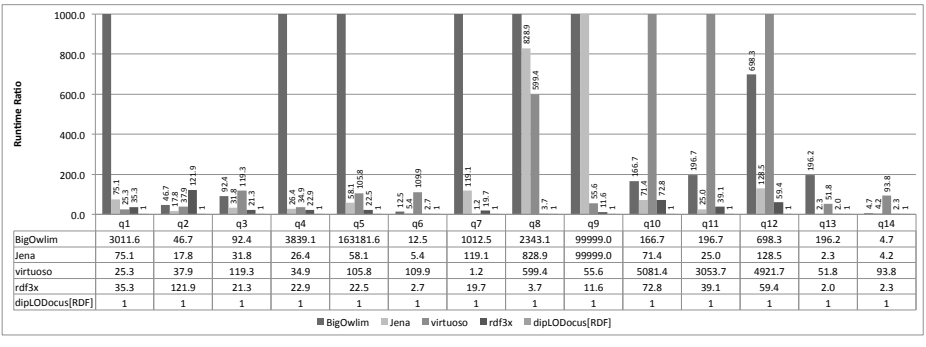
AllegroGraph: We used AllegroGraph RDFStore 4.2.1 AllegroGraph unfortunately poses some limits on the number of triples that can be stored for the free edition, such that we couldnt load the big data set. It also showed difficulty to deal with one query. For AllegroGraph, we prepared a SPARQL Python script using libraries supported by the vendor.

[10] `http://www.tpc.org/`



	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
AllegroGraph	752.3	945.4	180.7	58077.0	89156.6	84.6	4533.2	7780.9	99999.0	221209.1	942.5	5064.5	598.8	90.7
BigOwlim	3703.4	42.9	124.4	456.6	15396.6	13.5	384.1	266.3	99999.0	170.5	173.2	626.4	1237.4	14.7
Jena	82.8	18.1	52.6	27.7	54.5	3.3	46.0	83.0	99999.0	46.1	25.0	120.7	21.0	2.8
virtuoso	8.9	4.1	52.4	22.9	39.1	37.7	1.3	83.6	17.0	411.7	39.6	123.1	80.3	41.6
rdf3x	63.1	115.5	37.9	23.7	25.4	1.5	1.7	1.4	2.7	64.5	21.4	82.1	22.4	1.8
dipLODocus[RDF]	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 4. Runtime ratios for the 10 universities data set



	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14
BigOwlim	3011.6	46.7	92.4	3839.1	163181.6	12.5	1012.5	2343.1	99999.0	166.7	196.7	698.3	196.2	4.7
Jena	75.1	17.8	31.8	26.4	58.1	5.4	119.1	828.9	99999.0	71.4	25.0	128.5	2.3	4.2
virtuoso	25.3	37.9	119.3	34.9	105.8	109.9	1.2	599.4	55.6	5081.4	3053.7	4921.7	51.8	93.8
rdf3x	35.3	121.9	21.3	22.9	22.5	2.7	19.7	3.7	11.6	72.8	39.1	59.4	2.0	2.3
dipLODocus[RDF]	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 5. Runtime ratios for the 100 universities data set

BigOWLIM: We used BigOWLIM 3.5.3436. OWLIM provides us with a java application to run the LUBM benchmark, so we used it directly for our tests.

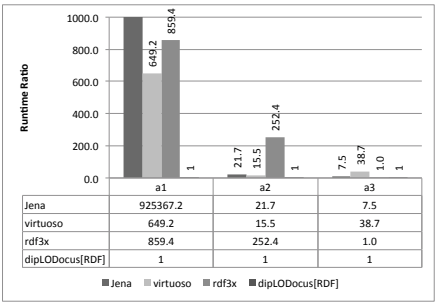
Jena: We used Jena-2.6.4 and the TDB-0.8.10 storage component. We created the database by using the "tdbloader" provided by Jena. We created a Java application to run and measure the execution time of each query.

Virtuoso: We used Virtuoso Open-Source Edition 6.1.3. Virtuoso supports ODBC connections, and we prepared a Python script using the PyODBC library for our queries.

RDF-3X: We used RDF-3X 0.3.5. For this system, we converted our dataset to NTriples/Turtle. We also hacked the system to measure the execution time of the queries only, without taking into account the initialization of the database and turning off the print-outs.

6.6 Results

Relative execution times for all queries and all systems are given below, in Figure 4 for 10 universities and in Figure 5 for 100 universities. Results are



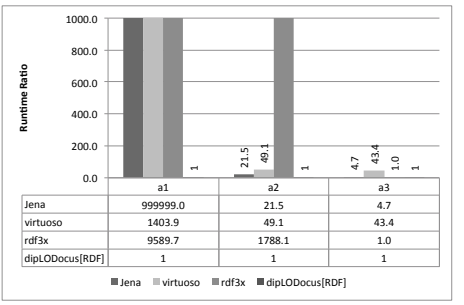


Fig. 6. Runtime ratios for 10 (left) and 100 (right) universities for the analytic/aggregate queries

10 UNI -- Query Execution Time [s]

		dipLODocus	AllegroGrap	BigOwlim	virtuoso	rdf3x	Jena
q1	AVG	1.45E-05	1.09E-02	5.37E-02	1.29E-04	9.14E-04	1.20E-03
	CI	6.47E-08	4.81E-05	6.27E-05	1.00E-06	2.23E-07	7.93E-06
q2	AVG	1.21E-02	1.14E+01	5.19E-01	4.96E-02	1.40E+00	2.19E-01
	CI	5.63E-05	2.96E-03	9.04E-04	4.72E-05	1.85E-05	1.42E-04
q3	AVG	2.09E-05	3.78E-03	2.60E-03	1.10E-03	7.91E-04	1.10E-03
	CI	9.57E-08	8.77E-06	1.32E-05	4.00E-06	1.11E-06	5.95E-06
q4	AVG	7.95E-05	4.62E+00	3.63E-02	1.82E-03	1.89E-03	2.20E-03
	CI	3.17E-07	8.49E-04	1.18E-04	1.04E-06	2.22E-07	7.93E-06
q5	AVG	5.32E-05	4.74E+00	8.19E-01	2.08E-03	1.35E-03	2.90E-03
	CI	2.56E-07	1.11E-03	2.89E-04	1.71E-05	3.89E-08	1.07E-05
q6	AVG	1.65E-02	1.40E+00	2.23E-01	6.22E-01	2.51E-02	5.52E-02
	CI	8.65E-05	3.93E-04	8.00E-04	1.58E-03	8.81E-06	2.70E-04
q7	AVG	1.22E-03	7.03E+01	5.96E+00	1.55E-03	4.82E-03	7.14E-01
	CI	3.21E-07	1.12E-02	2.18E-03	2.10E-06	5.04E-05	2.25E-04
q8	AVG	6.54E-03	5.09E+01	1.74E+00	5.47E-01	8.94E-03	5.43E-01
	CI	2.21E-05	8.28E-03	7.10E-04	1.09E-03	1.57E-06	1.47E-03
q9	AVG	6.74E-02	longer than	longer than	1.14E+00	1.83E-01	longer than
	CI	1.98E-06	two hours	two hours	4.38E-03	9.06E-05	two hours
q10	AVG	2.17E-05	4.80E+00	3.70E-03	8.93E-03	1.40E-03	1.00E-03
	CI	7.32E-08	1.15E-03	9.09E-06	7.49E-06	1.80E-06	0.00E+00
q11	AVG	6.41E-05	6.04E-02	1.11E-02	2.54E-03	1.37E-03	1.60E-03
	CI	2.32E-07	5.08E-04	5.95E-06	1.76E-05	3.25E-07	1.32E-05
q12	AVG	1.74E-05	8.81E-02	1.09E-02	2.14E-03	1.43E-03	2.10E-03
	CI	5.19E-08	8.05E-05	5.95E-06	1.93E-06	2.70E-07	5.95E-06
q13	AVG	4.76E-05	2.85E-02	5.89E-02	3.82E-03	1.06E-03	1.00E-03
	CI	1.18E-07	8.19E-05	5.71E-05	8.27E-07	7.82E-08	0.00E+00
q14	AVG	1.29E-02	1.17E+00	1.90E-01	5.37E-01	2.28E-02	3.62E-02
	CI	6.00E-05	3.79E-04	2.17E-04	1.91E-03	9.93E-06	1.82E-04
a1	AVG	1.16E-03	not run	not run	7.50E-01	9.93E-01	1.07E+03
	CI	2.24E-06			4.90E-04	3.90E-03	3.42E-02
a2	AVG	5.07E-05	not run	not run	7.85E-04	1.28E-02	1.10E-03
	CI	1.72E-07			4.62E-06	1.25E-05	5.95E-06
a3	AVG	1.07E-02	not run	not run	4.13E-01	1.10E-02	8.01E-02
	CI	1.57E-07			2.30E-03	2.49E-06	7.00E-04

100 UNI -- Query Execution Time [s]

		dipLODocus	BigOwlim	virtuoso	rdf3x	Jena
q1	AVG	1.73E-05	5.21E-02	4.38E-04	6.10E-04	1.30E-03
	CI	5.17E-08	5.35E-05	5.88E-07	3.53E-07	9.09E-06
q2	AVG	1.27E-01	5.94E+00	4.83E+00	1.55E+01	2.27E+00
	CI	6.41E-05	3.79E-03	8.82E-04	1.52E-04	4.09E-04
q3	AVG	3.14E-05	2.90E-03	3.75E-03	6.68E-04	1.00E-03
	CI	2.54E-08	1.07E-05	6.01E-07	1.56E-07	0.00E+00
q4	AVG	8.33E-05	3.20E-01	2.91E-03	1.90E-03	2.20E-03
	CI	7.15E-08	5.44E-04	3.32E-06	1.29E-07	7.93E-06
q5	AVG	5.34E-05	8.71E+00	5.65E-03	1.20E-03	3.10E-03
	CI	6.16E-08	2.28E-03	4.37E-06	1.61E-07	2.07E-05
q6	AVG	1.42E-01	1.77E+00	1.56E+01	3.77E-01	7.61E-01
	CI	1.64E-05	1.15E-03	1.25E-02	5.24E-05	7.28E-03
q7	AVG	2.63E-03	6.34E+01	3.11E-03	5.18E-02	7.46E+00
	CI	3.39E-05	1.55E-02	2.09E-06	9.53E-04	1.54E-03
q8	AVG	6.34E-03	1.49E+01	3.80E+00	2.36E-02	5.26E+00
	CI	1.71E-06	3.93E-03	7.60E-04	6.69E-06	5.19E-03
q9	AVG	2.61E-01	longer than	1.45E+01	3.01E+00	longer than
	CI	2.29E-05	two hours	1.92E-03	1.07E-03	two hours
q10	AVG	1.68E-05	2.80E-03	8.54E-02	1.22E-03	1.20E-03
	CI	1.19E-08	7.93E-06	3.59E-06	1.49E-07	7.93E-06
q11	AVG	6.00E-05	1.18E-02	1.83E-01	2.35E-03	1.50E-03
	CI	1.54E-08	7.93E-06	4.72E-05	7.76E-07	9.91E-06
q12	AVG	1.79E-05	1.25E-02	8.81E-02	1.06E-03	2.30E-03
	CI	5.95E-09	3.68E-05	8.23E-05	1.06E-06	9.09E-06
q13	AVG	5.62E-04	1.10E-01	2.91E-02	1.11E-03	1.30E-03
	CI	1.99E-08	2.39E-04	5.56E-05	1.01E-07	9.09E-06
q14	AVG	1.41E-01	6.68E-01	1.33E+01	3.27E-01	5.98E-01
	CI	2.18E-06	1.21E-03	6.99E-03	1.30E-04	6.59E-03
a1	AVG	1.04E-02	not run	1.45E+01	9.93E+01	longer than
	CI	1.22E-07		6.52E-04	1.05E-03	two hours
a2	AVG	6.50E-05	not run	3.19E-03	1.16E-01	1.40E-03
	CI	1.54E-08		2.35E-06	1.21E-04	9.71E-06
a3	AVG	1.55E-01	not run	6.72E+00	1.49E-01	7.25E-01
	CI	2.65E-07		1.94E-03	3.61E-05	2.88E-03

	dipLODocus	AllegroGrap	BigOwlim	virtuoso	rdf3x	Jena
Load Time	31s	13s	50s	88s	16s	98s
size	87MB	696MB	209MB	140MB	66MB	118MB

	dipLODocus	BigOwlim	virtuoso	rdf3x	Jena
Load Time	427s	748s	914s	214s	1146s
size on disk	913MB	2012MB	772MB	694MB	1245MB

Fig. 7. Absolute query execution and load times [s], plus size of the databases on disk for both data sets

given as runtime ratios, with dipLODocus$_{[\mathrm{RDF}]}$ taken as a basis for ratio 1.0 (i.e., a bar indicating 752.3 means that the execution time of that query on that system was 752.3 times slower than the dipLODocus$_{[\mathrm{RDF}]}$ execution). Figure 6 gives relative execution times for analytics executed on a selection of the fastest systems. Absolute times with confidence intervals at 95%, database sizes on disk and load times are given in Figures 7 for both datasets.

We observe that dipLODocus$_{[\mathrm{RDF}]}$ is generally speaking very fast, both for bulk inserts, for LUBM queries and especially for analytic queries. dipLODocus$_{[\mathrm{RDF}]}$ is not the fastest system for inserts, and produces slightly larger databases on disk than some other systems (like RDF-3x), but performs overall very-well

for all queries. Our system is on average 30 times faster than the fastest RDF data management system we have considered (i.e., RDF-3X) for LUBM queries, and on average 350 times faster than the fastest system (Virtuoso) on analytic queries. Is is also very scalable (both bulk insert and query processing scale gracefully from 10 to 100 universities).

7 Conclusions

In this paper, we have described dipLODocus$_{[RDF]}$, a new RDF management system based on a hybrid storage model and RDF templates to execute various kinds of queries very efficiently. In our performance evaluation, dipLODocus$_{[RDF]}$ is on average 30 times faster than the fastest RDF data management system we have considered (i.e., RDF-3X) on LUBM queries, and on average 350 times faster than the fastest system we have considered on analytic queries. More importantly, dipLODocus$_{[RDF]}$ is the only system to consistently show low processing times for *all* the queries we have considered (i.e., our system is the only system being able to answer any of the queries we considered in less than one second), thus making it an extremely versatile RDF management system capable of efficiently supporting both short and long-tail queries in real deployments.

This impressive performance can be explained by several salient features of our system, including: its extremely compact structures based on molecule templates to store data, its redundant structures to optimize different types of operations, its very efficient ways of coping with disk and memory reads (avoiding seeks and memory jumps as much as possible since they are extremely expensive on modern machines), and its way of materializing various joins in all its data structures. This performance is counterbalanced by relatively complex and expensive updates and inserts, which can however be optimized if considered in bulk.

In the near future, we plan to work on cleaning, proof-testing, and extending our code base to deliver an open-source release of our system as soon as possible[11]. We also have longer-term research plans for dipLODocus$_{[RDF]}$; our next research efforts will revolve around parallelizing many of the operations in the system, to take advantage of multi-core architectures on one hand, and large cluster of commodity machines on the other hand. Also, we plan to work on the automated database design problem in order to automatically suggest sets of optimal root nodes to the database administrator given some sample input data and an approximate query workload.

Acknowledgment. This work is supported by the Swiss National Science Foundation under grant number PP00P2_128459.

References

1. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-grid: A self-organizing structured p2p system. ACM SIGMOD Record 32(3) (2003)

[11] visit `http://diuf.unifr.ch/xi/diplodocus` for updates.

2. Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Van Pelt, T.: GridVine: Building Internet-Scale Semantic Overlay Networks. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 107–121. Springer, Heidelberg (2004)
3. Agrawal, S., Chaudhuri, S., Narasayya, V.: Automated selection of materialized views and indexes in SQL databases. In: International Conference on Very Large Data Bases, VLDB (2000)
4. Atre, M., Chaoji, V., Weaver, J., Williamss, G.: Bitmat: An in-core rdf graph store for join query processing. In: Rensselaer Polytechnic Institute Technical Report (2009)
5. Broekstra, J., Kampman, A., Harmelen, F.V.: Sesame: An architecture for storing and querying rdf data and schema information. In: Semantics for the WWW. MIT Press (2001)
6. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient sql-based rdf querying scheme. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 1216–1227. VLDB Endowment (2005)
7. Cudré-Mauroux, P., Agarwal, S., Aberer, K.: Gridvine: An infrastructure for peer information management. IEEE Internet Computing 11(5) (2007)
8. Cudré-Mauroux, P., Lim, K., Simakov, R., Soroush, E., Velikhov, P., Wang, D.L., Balazinska, M., Becla, J., DeWitt, D., Heath, B., Maier, D., Madden, S., Patel, J.M., Stonebraker, M., Zdonik, S.: A Demonstration of SciDB: A Science-Oriented DBMS. Proceedings of the VLDB Endowment (PVLDB) 2(2), 1534–1537 (2009)
9. Cudré-Mauroux, P., Wu, E., Madden, S.: The Case for RodentStore, an Adaptive, Declarative Storage System. In: Biennial Conference on Innovative Data Systems Research, CIDR (2009)
10. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (2008)
11. Demartini, G., Enchev, I., Gapany, J., Cudré-Maurox, P.: BowlognaBench—Benchmarking RDF Analytics. In: SIMPDA 2011: First International Symposium on Process Data (2011)
12. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudré-Mauroux, P., Madden, S.: Hyrise - a main memory hybrid storage engine. PVLDB 4(2), 105–116 (2010)
13. Guo, Y., Pan, Z., Heflin, J.: An Evaluation of Knowledge Base Systems for Large OWL Datasets. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 274–288. Springer, Heidelberg (2004)
14. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semant. 3, 158–182 (2005)
15. Haslhofer, B., Roochi, E.M., Schandl, B., Zander, S.: Europeana RDF Store Report. University of Vienna, Technical Report (2011), `http://eprints.cs.univie.ac.at/2833/1/europeana_ts_report.pdf`
16. Liu, B., Hu, B.: An evaluation of rdf storage systems for large data applications. In: First International Conference on Semantics, Knowledge and Grid, SKG 2005, p. 59 (November 2005)
17. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment (PVLDB) 1(1), 647–659 (2008)
18. Prud'hommeaux, E., Seaborne van Harmelen, A. (eds.): SPARQL Query Language for RDF. W3C Candidate Recommendation (April 2006), `http://www.w3.org/TR/rdf-sparql-query/`
19. Ramamurthy, R., DeWitt, D.J., Su, Q.: A case for fractured mirrors. In: CAiSE 2002 and VLDB 2002. VLDB Endowment, pp. 430–441 (2002)

20. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S.R., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: A Column Oriented DBMS. In: International Conference on Very Large Data Bases, VLDB (2005)
21. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceeding of the VLDB Endowment (PVLDB) 1(1), 1008–1019 (2008)
22. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D.: Efficient rdf storage and retrieval in jena2. In: SWDB 2003, pp. 131–150 (2003)
23. Yan, Y., Wang, C., Zhou, A., Qian, W., Ma, L., Pan, Y.: Efficient indices using graph partitioning in rdf triple stores. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, pp. 1263–1266. IEEE Computer Society, Washington, DC, USA (2009)
24. Zou, L., Mo, J., Chen, L., Oezsu, M.T., Zhao, D.: gstore: Answering sparql queries via subgraph matching. PVLDB 4(8) (2011)