DREAM in Action: A Distributed and Adaptive RDF System on the Cloud

Aisha Hasan and Mohammad Hammoud
Carnegie Mellon University in Qatar
Education City, Doha, Qatar
{aishah, mhhamoud}@cmu.edu

Reza Nouri and Sherif Sakr
University of New South Wales
Sydney, NSW 2052 Australia
{snouri, ssakr}@cse.unsw.edu.au

ABSTRACT

RDF and SPARQL query language are gaining wide popularity and acceptance. This demonstration paper presents DREAM, a hybrid RDF system, which combines the advantages and averts the disadvantages of the centralized and distributed RDF schemes. In particular, DREAM avoids partitioning RDF datasets and reversely partitions SPARQL queries. By not partitioning datasets, DREAM offers a general paradigm for different types of pattern matching queries and entirely precludes intermediate data shuffling (only auxiliary data are shuffled). By partitioning only queries, DREAM suggests an adaptive scheme, which runs queries on different numbers of machines depending on their complexities. DREAM achieves these goals and significantly outperforms related systems via employing a novel graph-based, rule-oriented query planner and a new cost model. This paper proposes demonstrating DREAM live over the cloud using a friendly graphical user interface (GUI). The GUI allows participants to execute and visualize pre-defined and user-defined (which can be written by participants on-the-fly) SPARQL queries over various real-world and synthetic RDF datasets. Furthermore, participants can empirically compare and contrast DREAM against three state-of-the-art RDF systems.

1. INTRODUCTION

RDF is designed to flexibly model schema-free information for the Semantic Web. Specifically, it structures data items as *triples* of the form (S, P, O), where S stands for subject, P for predicate and O for object. A triple represents a relationship between S and O captured by P. As such, a collection of triples can be modeled as a directed graph, with vertices denoting subjects and objects, and edges indicating predicates.

RDF triples can be stored using different storage organizations, including relational tables [6], bitmap matrices [4] and native graph formats [8], among others. In practice, all RDF repositories can be searched using SPARQL queries that are composed of *triple patterns*. A triple pattern is much like a triple, except that S, P and/or O can be variables or literals (S, P and O in triples are only literals). Similar to triples, triple patterns can be modeled as directed graphs. Accordingly, resolving a SPARQL query can be framed as a sub-graph pattern matching problem [11].

WWW'16 Companion, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4144-8/16/04.
http://dx.doi.org/10.1145/2872518.2901923.

The wide adoption of the RDF data model calls for efficient and scalable RDF schemes. In response to this call, many systems were proposed, adopting either a *centralized* or a *distributed* paradigm. Centralized systems [2, 6, 12] store RDF datasets unsliced and do not partition[1] SPARQL queries. Their fundamental merit is that they do not incur any network traffic. However, they are typically bound by the CPU and memory resources of a single machine. Yet, a single machine with a *modern* disk can still fit any current RDF dataset (i.e., a dataset with millions or billions of triples), but will result in severe thrashing to main memory and frequent accesses to disk [7]. Evidently, this can lead to unacceptable performance degradation.

In an attempt to overcome the problems of centralized schemes, distributed systems [11, 13, 14] were suggested. In particular, these systems partition input RDF datasets among clustered machines, thus benefiting from larger aggregate memories and higher CPU power. Nonetheless, due to data partitioning, they induce (high) communication overhead when satisfying (complex) SPARQL queries, especially if the queries span multiple disjoint partitions.

In this demonstration, we present DREAM [7], a ***D***istributed ***R***DF ***E***ngine with ***A***daptive query planner and ***M***inimal communication. DREAM adopts a hybrid paradigm, which retains the benefits of the classical centralized and distributed schemes, and averts their drawbacks. More precisely, DREAM stores a given RDF dataset intact at each cluster machine (similar to centralized systems) and executes SPARQL queries across machines (similar to distributed systems), after applying a new *query partitioning* algorithm. By the virtue of this new paradigm, DREAM can: (1) totally eliminate data partitioning, which is theoretically NP-hard, and subsequently offer a one-size-fits-all model for different pattern matching queries (e.g., star-like and chained), (2) considerably reduce network traffic by avoiding data shuffling and communicating only *auxiliary*[2] data across machines, and (3) *adaptively* run any SPARQL query in a centralized or a distributed fashion, depending on its complexity. This flexibility is inherently provided by the unsliced data kept at each machine, which enables centralized execution when needed. Furthermore, it is effectively realized through a novel I/O-aware, rule-based query planner.

To this end, we propose demonstrating the full features of DREAM over cloud using a comprehensive, yet friendly graphical user interface (GUI). Through this GUI, participants will be able to test and validate DREAM via executing standard and new SPARQL queries over real-world and synthetic RDF datasets. In addition, they will be able to visualize in real-time how DREAM satisfies any

[1]By *partitioning* a SPARQL query, Q, we mean decomposing Q into multiple sub-queries *and* distributing them across clustered machines. Clearly, this is not an option for centralized systems.
[2]Auxiliary data denote minimal control messages and triple identifiers (i.e., not actual triples).

query, starting from receiving the query, generating a corresponding near-optimal plan, executing the plan, and outputting final actual and quantitative results. Lastly, they will have the opportunity to run three related centralized and distributed RDF systems [12, 11, 13], and compare their performance and network results versus DREAM. The details of the DREAM[3] project can be found at [1]. We provide a brief overview of DREAM in Section 2 and discuss our proposed demonstration scenarios in Section 3.

2. DREAM

2.1 Architecture Overview

DREAM adopts a master-slave architecture, with a single machine acting as a *master* and multiple clustered machines serving as *slaves*. The master hosts the system's intelligence (or the query planner). The query planner decides how the resolution of a query shall proceed. Each slave incorporates a centralized RDF store, which is responsible for maintaining an input RDF dataset and processing SPARQL (sub-)queries, as delegated by the master. Note that DREAM does not stipulate a specific RDF storage model (i.e., any centralized relational-based [2, 6] or graph-based [3, 5, 8] store can be utilized).

When a client submits a SPARQL query, Q, the query planner first transforms Q into a query graph, G. Next, the query planner produces a near-optimal *graph plan*, G_P, as a set of sub-graphs $\{SG_1, ..., SG_M\}$, where M is less than or equal to the number of slaves. Subsequently, the master delegates each sub-graph SG_i ($1 \leq i \leq M$) to a single slave, and all sub-graphs (if $M > 1$) are run in parallel (if M evaluates to 1, only 1 machine is used). At a slave, a sub-graph can be further optimized by the RDF store's query optimizer (if any). During execution, slaves exchange intermediate *auxiliary data*, join intermediate result sets and produce the final result. Next, we explain how the query planner produces a near-optimal graph plan.

2.2 An Adaptive Query Planner

2.2.1 Creating Query Graphs

The precursor step to partitioning a SPARQL query, Q, in DREAM is to produce its corresponding *query graph*. More formally, the query planner models Q as a directed graph, G. G is defined as $G = \{V, E\}$, where V and E are the sets of vertices and edges, respectively. Vertices in V and edges in E represent subjects/objects and predicates of triple patterns, respectively. For example, Fig. 1 portrays a SPARQL query $Q1$ and its corresponding directed graph $G1$. $Q1$ consists of five *basic sub-queries* $\{q_1, q_2, q_3, q_4, q_5\}$, which are reflected in $G1$ as *basic sub-graphs* $\{g_1, g_2, g_3, g_4, g_5\}$. A *basic sub-query* is a single triple pattern, or the smallest possible query structure. A *basic sub-graph* is the smallest possible graph structure, which corresponds to a basic sub-query. At the end of this step, the query planner begins the query graph partitioning stage.

2.2.2 Partitioning Query Graphs

In order to construct a near-optimal graph plan for a query graph G, the query planner begins by locating the vertices in G with degrees greater than 1. For instance, the degree of vertex `?Tournament` in Fig. 1 (b) is 3 (i.e., out-degree is 2 and in-degree is 1). We call such a vertex a ***join vertex***. After identifying join vertices, the query planner creates many empty sets S_{JV}s for every join vertex, JV, and populates them with specific basic sub-graphs from G, using a *rule-based strategy* (to be discussed shortly). Eventually, only one set, S_{JV}, for each join vertex will be selected and executed at a slave.

Prior to discussing how the query planner populates each set S_{JV} with sub-graphs, we classify basic sub-graphs as either *exclusive* or *shared*. An exclusive basic sub-graph is a sub-graph with exactly one join vertex, while a shared basic sub-graph is a sub-graph with two join vertices (recall that any basic sub-graph has a maximum of two vertices). For example, g_1 in Fig. 1 (b) is an exclusive basic sub-graph, while g_2 is a shared one. The query planner walks through the directed graph G as if it is undirected (starting at a random vertex), locates exclusive and shared basic sub-graphs and assigns them to sets S_{JV}s according to the following four rules:

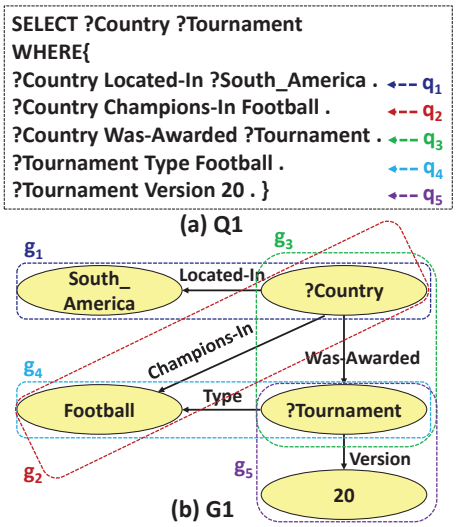

Figure 1: A SPARQL query, $Q1$, and its corresponding query graph, $G1$. $\{q_1, q_2, q_3, q_4, q_5\}$ are basic sub-queries and $\{g_1, g_2, g_3, g_4, g_5\}$ are their respective basic sub-graphs.

Rule 1: A basic sub-graph, g_i, can be assigned to a set S_{JV} if g_i is directly connected to the join vertex JV. For instance, the exclusive basic sub-graph g_1 in Fig. 1 (b) can be assigned to set $S_{?Country}$, but not to set $S_{?Tournament}$, as it is directly connected to `?Country` but not to `?Tournament`.

Rule 2: An exclusive basic sub-graph, g_i, which is directly connected to join vertex JV, *should* be assigned to *only* set S_{JV}. For example, the exclusive basic sub-graph g_1 in Fig. 1 (b) should be assigned to only set $S_{?Country}$ (hence, the name *exclusive*).

Rule 3: A shared basic sub-graph, g_i, which is directly connected to join vertices $JV1$ and $JV2$, *should* be assigned to *only* set S_{JV1} or set S_{JV2} or both. For instance, the shared basic sub-graph g_3 in Fig. 1 (b) should be assigned only to set $S_{?Country}$ or set $S_{?Tournament}$ or both (hence, the name *shared*).

Rule 4: Any set S_{JV} *should* include *at least two directly connected basic sub-graphs*, referred to as TD-CONN. As an example of a TD-CONN, $\{g_1, g_2\}$ in Fig. 1 (b) form a TD-CONN, while $\{g_1, g_4\}$ do not.

For a discussion on the justifications and implications of these rules, please refer to [7]. To this end, Table 1 illustrates the resultant S_{JV}s of each JV in $G1$ (shown in Fig. 1(b)) after applying the above four rules.

Table 1: Possible sets of join vertices of $G1$ (Fig. 1).

Join Vertex	Possible Set(s)
`?Country`	$S_{?Country} = \{g_1, g_2\}$ or $\{g_1, g_3\}$ or $\{g_1, g_2, g_3\}$
`?Tournament`	$S_{?Tournament} = \{g_5, g_3\}$ or $\{g_5, g_4\}$ or $\{g_5, g_4, g_3\}$
`Football`	$S_{?Football} = \{g_2, g_4\}$

2.2.3 Generating Base Graph Plans

Having generated the sets, S_{JV}s, of every join vertex, JV, in a given query graph, G, the query planner is ready to enumerate all

[3]The complete source code of DREAM is publicly available on https://github.com/CMU-Q/DREAM.

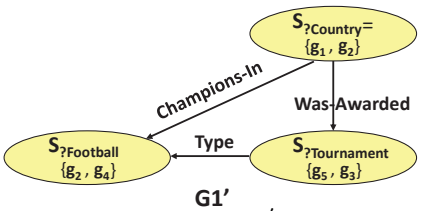

Figure 2: A possible graph plan, $G1'$, for the query graph, $G1$, in Fig. 1 (b).

possible graph plans, G_Ps, of G. As a first (and *basic*) step, we define a ***base graph plan***, G_P, as a directed graph consisting of exactly one S_{JV} from the sets, $S_{JV}s$, of *every* JV in G. As such, G_P incorporates a number of vertices that is equal to the number of join vertices in G. To exemplify, since $G1$ in Fig. 1(b) has 3 join vertices, its G_P will also have 3 vertices, each selected from a row in Table 1. Fig. 2 depicts one such G_P. Now, let us denote every S_{JV} in G_P as v'. Subsequently, any two vertices, v'_i and v'_j, in G_P, selected from the sets of join vertices JV_i and JV_j in G, will be connected by an edge, e'_{ij}, which corresponds to the edge, e, in G connecting JV_i and JV_j. Therefore, G_P is structurally identical to G but semantically different.

After generating all possible base graph plans, G_Ps, of G, the natural question that follows is: which of these graph plans should the query planner choose? The query planner employs a new cost model to estimate the I/O cost of each enumerated graph plan and, subsequently, selects the lowest-cost graph plan, G' (see [7] for details on our cost model). The ultimate goal of the query planner is to parallelize the execution of G by mapping each S_{JV} of a JV to a dedicated slave machine.

2.2.4 Generating Compact Graphs

As implied earlier, the number of join vertices in a query graph, G, dictates the number of machines for a generated lowest-cost base graph-plan, G'. However, some simple SPARQL queries might not need a distributed system whatsoever. In principle, what should dictate the number of machines for G' are the system resources (mainly memory) that G' requires, rather than G''s number of join vertices. Hence, to effectively execute G', we suggest examining the full continuum of potential numbers of machines, N, where $1 \quad N$ number of join vertices in G', and select N that will expectedly result in the best performance.

We realize our proposal by gradually *compacting* G', all the way until a single join vertex is obtained. Specifically, if the number of join vertices of G' is greater than one, we re-feed it to the query planner. The query planner, in turn, compacts G' (i.e., merges two neighboring join vertices and their respective sub-graphs) to produce a ***compact graph plan***. The compaction process continues until a graph plan with only a solo join vertex is attained. During this process, the query planner estimates the cost of every generated compact graph plan. Finally, the graph plan with the minimum estimated cost, say G^*, is selected and executed. This way, DREAM adaptively elects either a centralized or a distributed system with potentially different numbers of machines for different SPARQL queries.

2.3 Execution

Once the near-optimal graph plan, G^*, has been identified, the final task is to execute G^*. To do so, the query planner maps the set (which consists of a TD-CONN and zero or more sub-graphs) of each join vertex to a single slave. Consequently, all slaves run their sub-graphs in parallel, communicate intermediate auxiliary data (as dictated by the directed edges of G^*) and join intermediate result sets. At any slave machine, the received auxiliary data is used to locate the relevant triples from its local RDF store to proceed with its join(s). The final result is produced by a single slave machine and communicated back to the master.

2.4 Evaluation

Due to page constraints, we briefly indicate that we evaluated DREAM [7] on private and public clouds, and compared it against two state-of-the-art distributed RDF systems, Huang et *al.* [11] and H2RDF+ [13]. As for workloads, we utilized the two standard benchmark suites, YAGO2 [9] and LUBM [10]. On average, DREAM outperformed Huang et *al.* and H2RDF+ by 81% and 91%, respectively. Besides, DREAM reduced network traffic by averages of 16% and 13.4% versus Huang et *al.* and H2RDF+, respectively. Finally, we studied the scalability of DREAM on Amazon EC2 using large-scale datasets varying from 3 billion (or 700 GB) to 7 billion (or 1.2 TB) triples. As an outcome, we observed that DREAM scales very well with huge datasets. For more details on these experiments as well as other investigational studies, please refer to [7].

3. DEMONSTRATION

This demonstration aims at allowing our audience to have an enriching experience with DREAM through a friendly graphical user interface (GUI), which acts as a portal to different DREAM's features. In particular, via the GUI, the audience will be able to specify different parameters (e.g., queries, datasets, and execution modes), submit a query or a batch of queries (i.e., a job) to DREAM, visualize graph plans generated by DREAM's query planner, observe how a selected query plan is mapped to and executed on various cluster machines, inspect resultant processing times, and compare against some state-of-the-art centralized and distributed RDF systems, namely, RDF-3X [12], Huang et *al.* [11], and H2RDF+ [13]. We next elaborate on how this experience will be precisely attained.

3.1 Demonstration Setup

To pursue our demonstration, we will utilize a client machine and a cluster of eleven virtual machines (VMs) provisioned on a private cloud. Specifically, a client machine running our GUI will be available to the audience and connected to an 11-VM cluster on a private cloud at CMU in Qatar, wherein DREAM and the related schemes (i.e., RDF-3X, Huang et *al.*, and H2RDF+) will be installed and executed. At each VM, real-world (e.g., YAGO2 [9]) and synthetic (e.g., LUBM [10]) RDF datasets will be pre-loaded, ready to be queried by different SPARQL queries as described shortly.

3.2 Audience Interaction

Participants will be able to interact with DREAM through our GUI portrayed in Fig. 3. The GUI is split into three panes, *input parameters*, *query planning and processing*, and *output*.

3.2.1 Pane 1: Input Parameters

To begin with, participants can specify application- and engine-specific parameters. Specifically, application parameters involve selecting and running pre-defined and user-defined SPARQL queries. The pre-defined queries are the standard YAGO2 and LUBM queries. The user-defined queries are ones that participants can write and execute on-the-fly. After selecting or writing one or many SPARQL queries, a participant can specify engine parameters like triggering a single query or a batch of queries using either the *serial* or the *batch* execution mode of DREAM. The serial mode of DREAM allows running a solo query at time, while the batch mode permits executing a job comprising several queries using DREAM's random or greedy job scheduler [7]. Subsequent to setting parameters, users can preview a graphical-based form (i.e., the query graph, G) of every selected or written query. Such visual depictions of queries will allow users to easily identify *join vertices*, and *shared* and *exclusive* basic sub-graphs, which are leveraged by DREAM's query

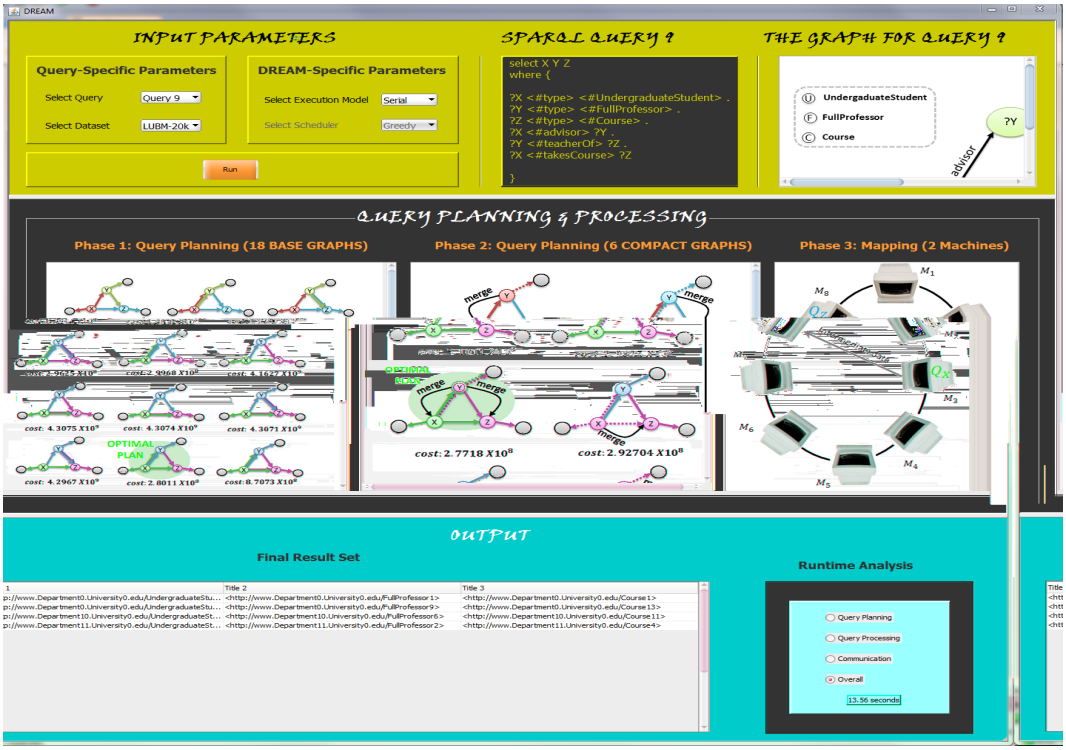

Figure 3: DREAM's GUI-based front-end tier running on a client machine. The GUI features DREAM's execution of a SPARQL query, $Q9$, derived from the standard LUBM benchmark[10].

planner to generate a near-optimal graph plan, G_P, of any G (see Section 2.2 for details).

3.2.2 *Pane 2: Query Planning and Processing*

After a participant selects or writes a SPARQL query, Q, she/he can submit Q to DREAM, wherein its query planner will be subsequently activated. Pane 2 illustrates the internal processing performed by the query planner. If the serial execution mode was selected, color-coded graph plans, generated by the query planner, will be displayed in real-time. The lowest-cost (or near-optimal) graph plan, G_P, will be then chosen using a novel cost model. Afterwards, G_P's constituent sub-graph(s) will be placed at one or many slave machine(s) (i.e., run as either centralized or distributed), depending on the complexity of G_P. If DREAM is run as distributed, the participant will be able to observe and validate the communication pattern(s) between them, which should at least respect the directionalities of edges in G_P. If the batch execution mode was selected, the mechanics of the specified job scheduler will be demonstrated, whereby the participant can view the query list, with queries getting enqueued and dequeued in real-time based on the scheduler's policy (e.g., greedy).

3.2.3 *Pane 3: Output*

This pane displays the final result set(s), coupled with a runtime breakdown. The runtime breakdown encompasses the time spent by DREAM on each of its major tasks: query planning, execution, and communication. This will enable the audience to thoughtfully assess the performance and network results of DREAM.

3.3 Comparisons with Related Schemes

Finally, the audience will be able to execute multiple state-of-the-art centralized and distributed RDF systems, namely RDF-3X [12], Huang *et al.* [11], and H2RDF+ [13]. All these systems will be deployed on the same 11-VM cluster of DREAM (for centralized RDF-3X, only one machine will be utilized), allowing participants to compare and contrast all the schemes using the same SPARQL queries and RDF datasets.

Acknowledgements

This publication was made possible by NPRP grant # 7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

4. REFERENCES

[1] Dream project - distributed rdf query engine. http://www.qatar.cmu.edu/~mhhammou/DREAM/index.html.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[3] R. Angles and C. Gutierrez. Querying rdf data from a graph database perspective. In *The Semantic Web: Research and Applications*. 2005.

[4] M. Atre, J. Srinivasan, and J. A. Hendler. Bitmat: A main-memory bit matrix of rdf triples for conjunctive triple pattern queries. In *ISWC (Posters & Demos)*, 2008.

[5] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *First Latin American of Web Congress*, 2003.

[6] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *SIGMOD*, 2013.

[7] M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *PVLDB*, 8(6), 2015.

[8] J. Hayes and C. Gutierrez. Bipartite graphs as intermediate model for rdf. In *ISWC*. 2004.

[9] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. De Melo, and G. Weikum. Yago2: exploring and querying world knowledge in time, space, context, and many languages. In *WWW*, 2011.

[10] http://swat.cse.lehigh.edu/projects/lubm/. The LUBM Benchmark.

[11] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11), 2011.

[12] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1), 2010.

[13] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H_2RDF+: an efficient data management system for big RDF graphs. In *SIGMOD*, 2014.

[14] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4), 2013.