DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication

Mohammad Hammoud [*], Dania Abed Rabbou [*], Reza Nouri [†],
Seyed-Mehdi-Reza Beheshti [†], Sherif Sakr [†‡]

[*]Carnegie Mellon University in Qatar, Education City, Doha, Qatar
{mhhamoud, dabedrab}@cmu.edu

[†]University of New South Wales, Sydney, NSW 2052 Australia
{snouri, sbeheshti, ssakr}@cse.unsw.edu.au

[‡]King Saud bin Abdulaziz University for Health Sciences, National Guard, Riyadh, Saudi Arabia

ABSTRACT

The Resource Description Framework (RDF) and SPARQL query language are gaining wide popularity and acceptance. In this paper, we present DREAM, a distributed and adaptive RDF system. As opposed to existing RDF systems, DREAM avoids partitioning RDF datasets and partitions only SPARQL queries. By not partitioning datasets, DREAM offers a general paradigm for different types of pattern matching queries, and entirely averts intermediate data shuffling (only auxiliary data are shuffled). Besides, by partitioning queries, DREAM presents an adaptive scheme, which automatically runs queries on various numbers of machines depending on their complexities. Hence, in essence DREAM combines the advantages of the state-of-the-art centralized and distributed RDF systems, whereby data communication is avoided and cluster resources are aggregated. Likewise, it precludes their disadvantages, wherein system resources are limited and communication overhead is typically hindering. DREAM achieves all its goals via employing a novel graph-based, rule-oriented query planner and a new cost model. We implemented DREAM and conducted comprehensive experiments on a private cluster and on the Amazon EC2 platform. Results show that DREAM can significantly outperform three related popular RDF systems.

1. INTRODUCTION

The Resource Description Framework (RDF) is gaining widespread momentum and acceptance among various fields, including science, bioinformatics, business intelligence and social networks, to mention a few. For instance, Semantic-Web-style ontologies and knowledge bases with millions of facts from DBpedia [5], Probase [39], Wikipedia [30] and Science Commons [40] are now publicly available. In addition, major search engines (e.g., Google, Bing and Yahoo!) are offering a better support for RDF [25, 41]. In short, Web content RDF-based management systems are proliferating in numerous communities all around the world [41].

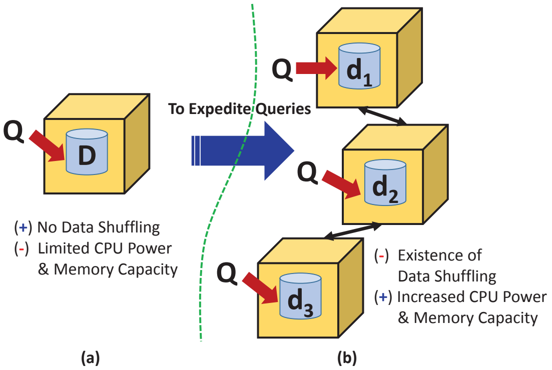

Q
D
(+) No Data Shuffling
(-) Limited CPU Power & Memory Capacity
To Expedite Queries
Q
d_1
Q
d_2
Q
d_3
(-) Existence of Data Shuffling
(+) Increased CPU Power & Memory Capacity
(a)
(b)

Figure 1: Current RDF systems (Dataset, $D = \{d_1, d_2, d_3\}$). (a) Centralized, and (b) Distributed schemes (SPARQL Query, Q, is not necessarily sent unsliced to each machine).

RDF is designed to flexibly model schema-free information for the Semantic Web [24, 40, 30]. It structures data items as *triples*, each of the form (S, P, O), where S stands for subject, P for predicate and O for object. A triple indicates a relationship between S and O captured by P. Consequently, a collection of triples can be modeled as a directed graph, with vertices denoting subjects and objects, and edges representing predicates. Triples can be stored using different storage organizations, including relational tables [9], bitmap matrices [4] and native graph formats [17], among others. All RDF stores 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, satisfying a SPARQL query is framed usually as a sub-graph pattern matching problem [25].

The wide adoption of the RDF data model calls for efficient and scalable RDF schemes. As a response to this call, many centralized RDF systems have already been suggested [1, 37, 9, 30, 40]. A main property of such systems is that they do not incur any communication overhead (i.e., they process all data locally). On the other hand, they remain limited by the capacities of computational resources of single machines (see Fig. 1 (a)). Specifically, with billions of RDF triples, tens or hundreds of gigabytes of main memory and a high-degree of parallelism will be required to rapidly satisfy the demands of *complex* SPARQL queries (i.e., queries with large numbers of triple patterns and joins) that are currently available only to high-end servers with steep prices [6, 27]. Nonetheless, a single machine with a modern disk can still fit any current RDF dataset (i.e., a dataset with

Proceedings of the VLDB Endowment, Vol. 8, No. 6
Copyright 2015 VLDB Endowment 2150-8097/15/02.

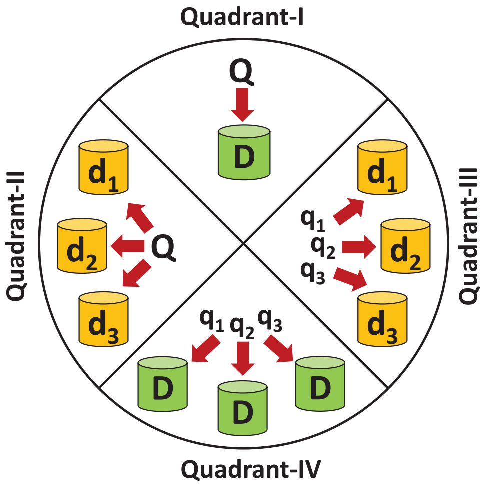


Figure 2: The four different paradigms for building RDF querying systems.

millions or billions of triples)[1], but will result in severe thrashing to main memory and frequent accesses to disk. Clearly, this can lead to unacceptable performance degradation. As a result, executing complex queries on a single machine might render infeasible, especially when the machine's main memory is dwarfed by the dataset size. To overcome this problem, recent work in literature proposed using distributed RDF systems rather than centralized ones [16, 34, 25, 41, 32].

With distributed systems, RDF data is typically partitioned among clustered machines using various partitioning algorithms such as hash or graph partitioning. Fig. 1 (b) depicts a distributed scheme, whereby a dataset, D, is divided into multiple partitions $\{d_1, d_2, d_3\}$ and placed at different machines. As opposed to centralized systems, distributed RDF systems are characterized by larger aggregate memory capacities and higher computational power. On the flip side, they might incur huge intermediate data shuffling when satisfying (complex) SPARQL queries, especially if the queries span multiple disjoint partitions. In principle, intermediate data shuffling can greatly degrade query performance. Hence, reducing intermediate data shuffling is becoming one of the major challenges in designing distributed RDF systems [25, 41].

As shown in Fig. 1, current state-of-the-art centralized and distributed systems promote entirely opposite paradigms for managing RDF data. More precisely, while centralized RDF systems avert intermediate network traffic altogether, they suffer from low computational power and limited memory capacities. In contrary, distributed RDF systems offer higher computational power and larger memory capacities, but incur (high) communication overhead. To our knowledge, RDF systems have not yet attempted to combine the benefits and preclude the shortcomings of both paradigms. To elaborate, our investigation of the RDF management problem suggests that RDF systems can be built in four different ways as portrayed in Fig. 2. All existing RDF systems lie under Quadrants I, II and III, wherein they either store an input RDF dataset, D, unsliced at a single machine and do not *partition*[2] a SPARQL query, Q (i.e., Quadrant-I or centralized), or partition D and/or Q (i.e., Quadrants II and/or III). Interestingly, there is no RDF system yet that falls under Quadrant-IV. With Quadrant-IV, D is maintained as is at each machine while Q is partitioned. Consequently, data shuffling can be completely avoided (i.e, each machine has all data) while computational power and memory capacities can be escalated, thus offering a hybrid paradigm between centralized and distributed schemes.

In this paper, we present DREAM, a ***D***istributed ***R***DF ***E***ngine with ***A***daptive query planner and ***M***inimal communication. DREAM is a Quadrant-IV citizen and the first in its breed. Accordingly, it retains the advantages of centralized and distributed RDF systems and obviates their disadvantages. DREAM stores a dataset intact at each cluster machine and employs a query planner that effectively partitions any given SPARQL query, Q. Particularly, the query planner transforms Q into a graph, G, decomposes G into many sets of sub-graphs, each with a basic two-level tree structure, and maps each set to a separate machine. Afterwards, all machines process their sets of sub-graphs in parallel and coordinate with each other to produce the final query result. No intermediate data is shuffled whatsoever and only minimal control messages and meta-data[3] (which we refer to both of them, henceforth, as ***auxiliary data***) are exchanged. To decide upon the number of sets (which dictates the number of machines) and their constituent sub-graphs (i.e., G's *graph plan*), the query planner enumerates various possibilities and selects a plan that will expectedly result in the lowest network and disk costs for G. This is achieved through utilizing a new cost model, which relies on RDF graph statistics. In the view of that, different numbers of machines for different query types are pursued by DREAM, hence, rendering it adaptive.

In particular, we summarize the main contributions of this paper as follows:

- We present DREAM, the first RDF system that attempts the Quadrant-IV paradigm shown in Fig. 2. Consequently, DREAM achieves minimal intermediate data communication (i.e., only auxiliary data are transferred).
- DREAM adaptively selects different numbers of machines for different SPARQL queries (depending on their complexities). This is accomplished via a novel query planner and a new cost model.
- We thoroughly evaluated DREAM using different benchmark suites over a private cluster and the Amazon EC2 platform. We further empirically compared DREAM against a popular centralized scheme [30] and two related distributed systems [25, 32]. Results show that DREAM can always adaptively select the best numbers of machines for the tested queries, and significantly outperform systems in [30], [25] and [32].

The rest of the paper is organized as follows. We motivate the case for the Quadrant-IV paradigm in Section 2. Details of DREAM, including its architecture and query planner, are discussed in Section 3. We present the evaluation methodology and results in Section 4. A summary of prior work is provided in Section 5 before we conclude in Section 6.

2. THE QUADRANT-IV PARADIGM

As pointed out in Section 1, with distributed systems, RDF data is typically partitioned across cluster machines using different partitioning methods like hash partitioning by subject, object or predicate, or graph partitioning by vertex. In reality, the choice of the partitioning algorithm largely impacts the volume of intermediate data

[1] A very large, Web-scale dataset that appeared recently in [32] includes 13.8 billion triples, which equates to *only* 2.5 TB. Physical hard drives with 4∼6 TB are inexpensively available nowadays (e.g., Seagate recently announced a 6 TB hard drive [23]). Let alone that Amazon EC2 currently offers instances with 24×2048 GB of disks. See Section 2 for more details.

[2] By partitioning a SPARQL query, Q, we mean decomposing Q into multiple sub-queries *and* distributing them across clustered machines. Many current RDF systems decompose queries within a single machine (for optimization reasons), but do not distribute constituent sub-queries across machines.

[3] In DREAM we use RDF-3X [30] at each slave machine and communicate only triple ids (i.e., meta-data) across machines. Locating triples using triple ids in RDF-3X is a straightforward process.

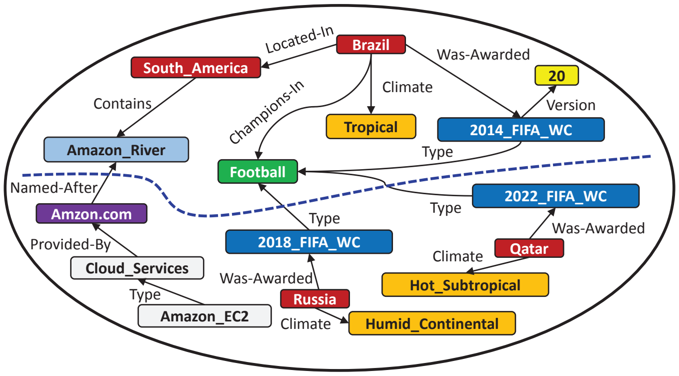


Figure 3: A sample RDF graph.

Q1
```
SELECT ?Country
WHERE{
?Country Located-In South_America .
?Country Champions-In Football . }
```

Q2
```
SELECT ?Country
WHERE{
?Country Located-In ?Continent .
?Continent Contains Amazon_River . }
```

Q3
```
SELECT ?Company
WHERE{
?Company Named-After Amazon_River .
Cloud-Services Provided-By ?Company . }
```

Q4
```
SELECT ?Company
WHERE{
?Company Named-After ?River .
?Continent Contains ?River .
Brazil Located-In ?Continent .
Cloud-Services Provided-By ?Company . }
```

Figure 4: Four sample SPARQL queries.

results. To exemplify, consider the sample RDF graph and SPARQL queries in Figures 3 and 4, respectively. For instance, $Q1$ in Figure 4 returns all the countries that are located in South America and are champions in Football (i.e., they won the FIFA World Cup at least once). If data is hash partitioned by subject (i.e., Country), it is guaranteed that all triples related to a particular country will be stored at the same machine. Thus, if $Q1$ is submitted to every machine, it will be satisfied in an embarrassingly parallel fashion and no communication of intermediate data will be incurred (of course, except at the very end upon aggregating all partial results). The answer of $Q1$ is {`Brazil`}. In fact, the type of $Q1$ is referred to as *star-shaped*, whereby many triple patterns are joined on a common column. Star-shaped queries are known to work quite well with hash partitioning [40, 25].

As opposed to $Q1$, $Q2$ in Fig. 4 returns all the countries that are located in the continent that contains the Amazon River. If we assume again that data is hash partitioned by subject, satisfying $Q2$ will likely result in intermediate data shuffling because the set of bindings of `?Country` and `?Continent` will be potentially hashed to different machines. The answer of $Q2$ is {`Brazil`}. With queries like $Q2$, where multiple triple patterns are joined on different columns, hash partitioning can greatly degrade performance due to communication overhead. We refer to this type of queries as *chained* queries.

To address the problem of chained queries, a recent work [25] suggested using graph partitioning by vertex instead of hash partitioning by subject, predicate or object. Specifically, the *METIS* partitioner [19] was employed in order for vertices that are nearby to be collocated on the same partition (except the ones at the boundary of the partition) and, subsequently, placed at the same machine. As an example, Fig. 3 demonstrates two possible partitions, separated by a dotted line. Clearly, $Q2$ in Fig. 4 can now be satisfied without causing any intermediate data shuffling. In contrast, $Q3$ in the same figure will still induce intermediate network traffic. $Q3$ returns all the companies that are named after the Amazon River and which provide Cloud Computing services. The answer of $Q3$ is {`Amazon.com`}. In principle, queries that span multiple partitions at different machines (like $Q3$) will always incur communication overhead and potentially degrade performance.

To reduce communication overhead, the work in [25] proposed an additional mechanism denoted as *n-hop guarantee*, which guarantees that any query involving n edges (or hops) from any vertex in a partition can be satisfied without communicating intermediate data. This is achieved via replicating vertices across partitions. Queries that are beyond n hops, however, still incur communication overhead and are handled using costly Hadoop jobs [20]. For instance, with 1-hop guarantee, $Q3$ in Fig. 4 can be executed without shuffling intermediate data. On the other hand, with 1-hop guarantee, $Q4$ in the same figure will still generate intermediate network traffic. $Q4$ returns all the companies that provide Cloud Computing services and are named after a river, which is found in the continent where Brazil is located. The answer to $Q4$ is {`Amazon.com`}.

Of course, n can be increased to 2 or more, and $Q4$ in Fig. 4 can be, accordingly, satisfied without transferring intermediate data. Nonetheless, several factors must be considered. First, RDF graphs with the power-law distribution[4] can cause major problems for graph partitioning and the n-hop guarantee mechanism. In particular, the degrees of vertices in such graphs can vary greatly and, thereby, result in severe skewed replications at different partitions when n-hop guarantee is applied. Obviously, this can create load imbalance and degrade query performance. As such, adopting an auxiliary mechanism which can equalize partitions might become necessary. Second, if queries are not partitioned (i.e., each query is sent to all cluster machines as is), a further mechanism for avoiding duplicate results must be incorporated. Third, the more the RDF graph is connected, the harder it is to partition. Hence, a strategy for reducing the connectivity of a given graph (e.g., removing triples whose predicate is *rdf:type*) shall be involved. Lastly, since star-shaped queries are common in SPARQL, graph partitioning is typically pursued by vertex and not by edge [25]. This entails employing a specific placement mechanism to decide which triple goes to which partition, during (or after) partitioning. Overall, a great deal of overhead will be ensued by partitioning and replicating the vertices of RDF graphs, even at medium-scale (i.e., for graphs with less than a billion vertices)[5].

In summary, we note two main points. First, partitioning (which is theoretically NP-hard [14]) and preprocessing RDF graphs can render extremely intricate and expensive. In essence, the larger and the more twisted the RDF graphs are, the harder graph partitioning algorithms turn out. Besides, the more complex SPARQL queries are, the less effective the n-hop guarantee mechanism becomes, especially when queries exceed n hops during execution (see Section 4 for empirical evidences). Second, as discussed through queries $Q1$, $Q2$, $Q3$ and $Q4$ in Fig. 4, different partitioning algorithms suit different queries. Thus, there is no *one-size-fits-all* partitioning algorithm. Indeed, any partitioning algorithm will result in intermediate data shuffling for some query workloads. Our objective is to entirely overcome these two problems and attain minimal intermediate data communication. A simple and effective approach to meet such an objective is to store each dataset unsliced at each cluster machine and, thereby: (1) preclude the complexity of partitioning and preprocessing algorithms altogether, and (2) offer a one-size-fits-all paradigm for all sorts of SPARQL queries (e.g., simple and complex

[4] In fact, many real-life RDF graphs are scale-free, whose vertex degrees follow the power-law distribution [41].

[5] To quantify the overhead of applying some of the above mechanisms, we measured the times the scheme at [25] takes to partition and apply the 2-hop guarantee for two standard datasets. We found that for an LUBM [22] dataset with 1 billion triples and a YAGO2 [18] dataset with 320 million triples, the scheme took 4.45 and 2.38 hours, respectively. With graphs at larger scales and n>2, these times are expected to grow exponentially.

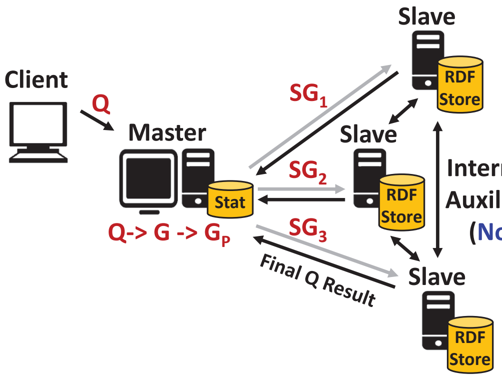


Figure 5: DREAM's architecture (Q = SPARQL Query; G = Q's Query Graph; G_P = $\{SG_1, SG_2, SG_3\}$ = G's Graph Plan).

with variances). We refer to this novel paradigm as Quadrant-IV (see Fig. 2).

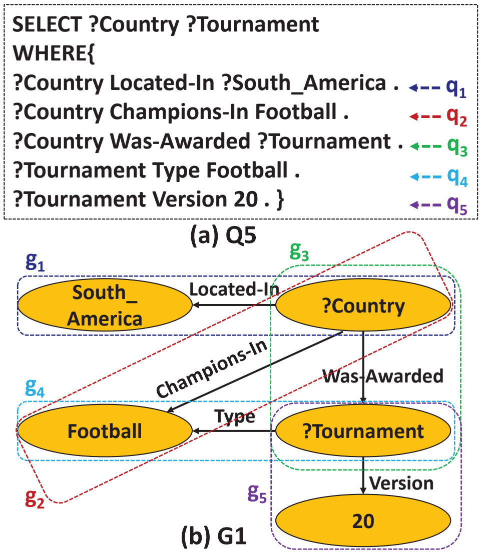
To this end, the authors in [27] observed that Big Graphs are not Big Data, indicating the feasibility of the Quadrant-IV paradigm. To put this in perspective, the largest RDF dataset that we know of nowadays consists of 13.6 billions of triples, which evaluates to only 2.5 TB [32]. Modern physical disks can fit 6 TB [23]. Furthermore, on Amazon EC2, users can provision EC2 instances with disk sizes of 24×2048 GB. Let alone that a user can attach multiple (e.g., 24) Amazon Elastic Block Storage (EBS) volumes to a single EC2 instance, each with a capacity of 1 TB. In Section 4, we test DREAM with 7 billion triples (or 1.2 TB) on Amazon EC2 using r3.2xlarge instance type and EBS volumes. Results demonstrate the effectiveness of DREAM. We next detail how DREAM implements the Quadrant-IV paradigm.

3. DREAM

3.1 Architecture

DREAM adopts a master-slave architecture shown in Fig. 5. Each slave machine can, in principle, encompass any centralized RDF store, including current relational-based [1, 9] and graph-based [3, 8, 17] stores (among others). Accordingly, DREAM offers a general-purpose scheme, whereby it does not impose any specific data model and can be easily tailored to incorporate any desired storage layout. Assuming an input RDF dataset, D, each slave machine in DREAM stores D unsliced, thus employing the Quadrant-IV paradigm. On the contrary, the master machine involves a query planner, detailed in Section 3.2. A client can submit any[6] SPARQL query, Q, to the master machine which, in turn, transforms it into a graph, G, and feeds it to the query planner. The query planner *partitions* G into a set of sub-graphs, G_P = $\{SG_1$, ..., $SG_M\}$, where M is less than or equal to the number of slave machines. Subsequently, the master places each sub-graph SG_i ($1 \leq i \leq M$) at a single slave machine, and all machines (if $M > 1$) are executed in parallel (M could evaluate to 1, and, thereby, only 1 machine will be used- see Section 3.2.5). During execution, slave machines exchange intermediate *auxiliary data*, join intermediate data and produce the final query result.

Since D is maintained as a whole at each machine, DREAM does not shuffle intermediate data whatsoever and only communicates identifiers of triples (i.e., auxiliary data). Besides, as slave machines can include any centralized RDF store (e.g., RDF-3X [30]), each sub-graph executed at each machine can be further optimized using the store's query optimizer (if any). Lastly, since any query graph, G, can be partitioned into many sub-graphs or kept as is, DREAM can run in a distributed or a centralized manner. This is dictated by the query planner which generates a *graph plan*, G_P (i.e., the set of partitioned sub-graphs), for each G that maximizes parallelism and minimizes network traffic. We next discuss how DREAM partitions G, generates and executes G_P, and produces G_P's final result.



```
SELECT ?Country ?Tournament
WHERE{
?Country Located-In ?South_America .  <--- q1
?Country Champions-In Football .      <--- q2
?Country Was-Awarded ?Tournament .    <--- q3
?Tournament Type Football .           <--- q4
?Tournament Version 20 . }            <--- q5
```

(a) Q5

(b) G1

Figure 6: A SPARQL query, $Q5$, 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.

3.2 Query Planner

3.2.1 Partitioning SPARQL Queries

As compared to existing RDF systems, DREAM partitions SPARQL queries rather than partitioning RDF datasets. This is achieved via firstly modeling any given SPARQL query, Q, as a directed graph, G. G is defined as $G = \{V, E\}$, where V is the set of vertices and E is the set of edges. Vertices in V and edges in E represent subjects/objects and predicates of triple patterns, respectively. For instance, Fig. 6 depicts a SPARQL query $Q5$ and its corresponding directed graph $G1$. $Q5$ 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. A basic sub-graph is modeled as two vertices connected by a directed edge. The two vertices represent the subject and the object of the respective basic sub-query, and the directed edge captures the relationship (or the predicate) between them.

After translating a SPARQL query Q to a directed graph G, the query planner partitions G into multiple sub-graphs. In particular, the query planner locates the vertices in G with degrees (i.e., in and out degrees) greater than 1. For instance, the degree of vertex `?Tournament` in Fig. 6 (b) is 3 (i.e., out-degree is 2 and in-degree is 1). We call a vertex with a degree greater than 1 a *join vertex*. As shown in Fig. 6 (b), in addition to `?Tournament`, vertices `?Country` and `Football` are join vertices. After locating join vertices, the query planner creates many empty sets S_{JV}s for every join vertex, JV, and populates them with specific sub-graphs from G, using a rule-based strategy (as discussed shortly). Eventually, only one set for each join vertex will be selected and mapped to a separate slave machine (see Section 3.2.4). Afterwards, all sets will

[6] By *any* in this DREAM version we mean SPARQL queries that involve only searches (i.e., no updates). Supporting updates is beyond the scope of this work and is set for future exploration. Indeed, all the related distributed RDF systems (see Section 5) do not support updates as well.

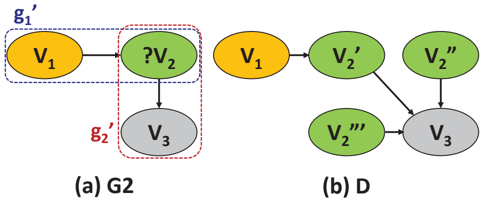


Figure 7: A query graph, $G2$, with a single TD-CONN $\{g_1$', g_2'$\}$ and a sample dataset, D. The result of running $G2$ over D is V_2'.

run in parallel and exchange auxiliary data as necessary to produce the final result of Q.

Before 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 basic sub-graph with only one join vertex, while a shared basic sub-graph is a basic sub-graph with two join vertices (recall that any basic sub-graph has a maximum of two vertices). For example, g_1 in Fig. 6 (b) is an exclusive basic sub-graph, while g_2 is a shared one. To this end, 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:

- **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. 6 (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. 6 (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. 6 (b) should be assigned to only 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. 6 (b) form a TD-CONN, while $\{g_1, g_4\}$ do not.

Clearly, *Rule 2* implies that an exclusive basic sub-graph should be assigned to exactly one set of a join vertex. In contrary, *Rule 3* entails that a shared basic sub-graph should be assigned to either one set of a join vertex or two sets of join vertices (not less, not more). Besides, *Rule 2* and *Rule 3* together guarantee that *all* basic sub-graphs will appear in the created sets of join vertices, thus *covering* the original query graph G[7]. Lastly, *Rule 4* suggests TD-CONN (and not a single basic sub-graph) as a mandatory unit structure for any set of a join vertex. In fact, traditional graph processing systems attempt to break graphs into large unit structures as well, when pursuing graph analytics (see for example [36]). Our rationale behind *Rule 4* is two-fold: (1) to avoid generating and communicating a large amount of superfluous intermediate results, and (2) to effectively prune the space of all possible sets of join vertices. We illustrate both objectives through examples.

Fig. 7 shows a query graph $G2$ with only one TD-CONN $\{g_1', g_2'\}$, and a sample dataset D (recall that an RDF dataset can also be modeled as a directed graph). The vertex $?v_2$ in G_2 is a variable, while v_1 and v_3 are literals. The bindings of $?v_2$ in D are v_2', v_2'', and v_2'''. If the basic sub-graphs g_1' and g_2' are segregated and executed at two different machines, v_2', v_2'' and v_2''' will be generated and communicated[8] (either all or some) as intermediate results. Successively, g_1' and g_2' will be joined and only v_2' will be returned as a final result. Therefore, the generation of v_2'' and v_2''' was superfluous since they were not part of the final result. On the flip side, if g_1' and g_2' are kept together (forming, thereby, a TD-CONN) and satisfied at the same machine, only v_2' will be output and no network traffic will be incurred. Of course, in essence we can map any SPARQL query as is (i.e., without partitioning) to only a solo machine and preclude communication altogether. However, this makes DREAM a pure centralized system. Our objective is rather to offer an *adaptive hybrid system*, which expedites query processing via leveraging maximal parallelism and minimal communication. This suggests running DREAM either centralized or distributed, with various numbers of machines, depending on the complexities of the given queries (more on this in Section 3.2.5).

Table 1: Possible sets of join vertices of $G1$ (Fig. 6 (b)).

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	$S_{?Football} = \{g_2, g_4\}$

Now to illustrate how TD-CONN serves in pruning the space of alternative sets of join vertices, Table 1 shows all the possible sets of join vertices produced by the query planner for the query graph $G1$ in Fig. 6 (b). As a specific example, for the join vertex ?Country in $G1$, the query planner generates three possible sets $\{g_1, g_2\}$, $\{g_1, g_3\}$, and $\{g_1, g_2, g_3\}$. First, g_4 and g_5 are not part of any set of ?Country, abiding by *Rule 1*. Second, g_1 is an exclusive basic sub-graph and, according to *Rule 2*, it *should* appear in every possible set of ?Country. Third, g_2 and g_3 are shared basic sub-graphs and, as dictated by *Rule 3*, *can* be included in any set of ?Country. Consequently, the query planner can assign only g_3, or only g_2, or both to any set of ?Country (as in $S_{?Country} = \{g_1, g_3\}$, $S_{?Country} = \{g_1, g_2\}$ or $S_{?Country} = \{g_1, g_2, g_3\}$). Finally, according to *Rule 4*, any set of ?Country should contain at least one TD-CONN. This is satisfied in $S_{?Country} = \{g_1, g_3\}$ through TD-CONN $\{g_1, g_3\}$, in $S_{?Country} = \{g_1, g_2\}$ through TD-CONN $\{g_1, g_2\}$, and in $S_{?Country} = \{g_1, g_2, g_3\}$ though TD-CONN $\{g_1, g_2\}$, or $\{g_1, g_3\}$, or $\{g_2, g_3\}$. In contrast, if TD-CONN is not obligatory (i.e., *Rule 4* does not exist), all combinations of g_1, g_2 and g_3 will morph into possible sets of ?Country (e.g., $\{g_1\}$ becomes a possible set). Clearly, this increases the space of alternative sets of join vertices and, potentially, escalates communication overhead (as shown in the example of Fig. 7)[9].

3.2.2 *Generating Graph Plans*

After creating all the sets of join vertices for a query graph G, the query planner generates a respective directed *set graph*, G'. G' is defined as $G' = \{V', E'\}$, where V' is the set of vertices and E' is the set of directed edges. Each vertex, v', in V' represents a set of join vertex S_{JV} created out of G, and each edge, e', in E' corresponds to an edge in G. Specifically, for every directed edge, e, in G connecting two join vertices $JV1$ and $JV2$, there will be a directed edge, e', in G' connecting two vertices, v_1' and v_2', which represent the sets, S_{JV1} and S_{JV2}, of join vertices $JV1$ and $JV2$.

[7] This applies to any input query graph of any type (e.g., star or chained) since it consists of only exclusive and/or shared basic sub-graphs, which are guaranteed by Rule 2 and Rule 3 to appear collectively in any generated sets of join vertices.

[8] More precisely, only auxiliary data will be communicated (see Section 3.2.2).

[9] As known, pruning the space of alternative query plans is common in traditional relational databases. We share this objective with relational databases and attempt to avoid rapid increase in possible sets as the number of join vertices is increased.

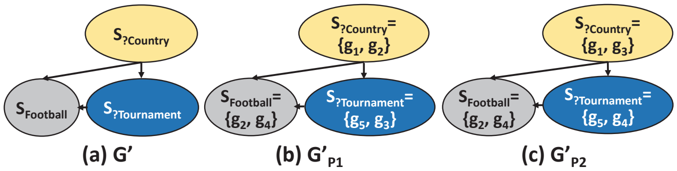


Figure 8: A set graph, G', which corresponds to the query graph, $G1$, in Fig. 6 (b). G_{P1}' and G_{P2}" are two possible graph plans for G'.

To exemplify, Fig. 8 (a) portrays a directed set graph G' for graph $G1$ in Fig. 6 (b). As shown in Table 1, the query planner creates three sets of join vertices out of $G1$, hence, G' contains three pertaining vertices, $S_{?Country}$, $S_{?Tournament}$ and $S_{Football}$. The directed edges between $S_{?Country}$ and $S_{Football}$, $S_{?Country}$ and $S_{?Tournament}$, and $S_{?Tournament}$ and $S_{Football}$ correspond to the directed edges between `?Country` and `Football`, `?Country` and `?Tournament`, and `?Tournament` and `Football` in $G1$.

After generating a directed set graph G' for a query graph G, the query planner constructs many *graph plans*, $G'_{P}s$ for G'. A graph plan G'_P is structurally identical to G', but with different sets of sub-graphs assigned to each vertex in G'_P. In particular, to construct G'_P, the query planner assigns to each vertex, S_{JV}, in G' (which corresponds to join vertex JV in G), a *unique* set of sub-graphs, from among many possible sets of sub-graphs. For instance, the query planner can assign $\{g_1, g_2\}$ or $\{g_1, g_3\}$ or $\{g_1, g_2, g_3\}$ from Table 1 to vertex $S_{?Country}$ in G' shown in Fig. 8 (a). Similarly, the query planner can assign $\{g_5, g_3\}$ or $\{g_5, g_4\}$ or $\{g_5, g_4, g_3\}$, and $\{g_2, g_4\}$ from Table 1 to vertices $S_{?Tournament}$ and $S_{Football}$ in G', respectively. Fig. 8 (b) and Fig. 8 (c) depict two possible graph plans, G'_{P1} and G'_{P2}, for G'. In theory, for k join vertices in G', the query planner can generate $n_1 \times n_2 \times ... \times n_k$ possible graph plans, where n_j ($1 \leq j \leq k$) is the number of possible sets of sub-graphs of join vertex j. For example, the join vertices, $S_{?Country}$, $S_{?Tournament}$ and $S_{Football}$ in G' have 3, 3 and 1 possible sets of sub-graphs, respectively (see Table 1). Accordingly, there are a total of 9 (i.e., $3 \times 3 \times 1$) possible graph plans for G'.

3.2.3 Executing a Selected Graph Plan

After enumerating many possible graph plans for a set graph G', the query planner chooses *only one* graph plan, G'_P, which is typically the *lowest-cost plan* for G'. We discuss in Section 3.2.4 how the query planner estimates costs of graph plans and selects G'_P. As for now, after picking G'_P, the query planner maps each of its vertices to a single slave machine in DREAM. Consecutively, all slave machines satisfy G'_P's vertices in parallel, communicate intermediate auxiliary data as implied by G'_P's directed edges, join intermediate data and produce G'_P's final result (done by only one slave machine). To illustrate, let us assume that G'_{P1} in Fig. 8 (b) is the lowest-cost plan of the set graph G' in Fig. 8 (a). Let us further assume that $S_{?Country} = \{g_1, g_2\}$, $S_{?Tournament} = \{g_5, g_3\}$ and $S_{Football} = \{g_2, g_4\}$ are mapped to slave machines, M_1, M_2 and M_3, respectively. As indicated by the directed edges in G'_{P1}, once M_1 is done with executing g_1 and g_2, it sends auxiliary data about its intermediate triples to M_2 and M_3. Next, M_2 and M_3 use the received auxiliary data to locate the relevant triples from their RDF stores and join them with their locally generated intermediate results. Finally, M_2 sends auxiliary data about its latest intermediate data to M_3, which, in turn, reads the corresponding triples, joins them with its most recent intermediate data, and outputs G'_{P1}'s final result.

3.2.4 Estimating Costs of Graph Plans

As discussed in Section 3.2.2, the query planner can generate multiple graph plans for any input query graph. The natural question that follows is: which of these graph plans should the query planner choose? As pointed out, the query planner estimates the cost of each possible graph plan and selects the one with the minimum cost. Before we delve into more details about how this is done, we define three types of vertices, *start-vertex*, *mid-vertex* and *end-vertex*. A start-vertex is a vertex with outgoing but no incoming edges. A mid-vertex is a vertex with incoming and outgoing edges. An end-vertex is a vertex with incoming but no outgoing edges. For instance, in G'_{P1} (Fig. 8 (b)), $S_{?Country}$ is a start-vertex, $S_{?Tournament}$ is a mid-vertex and $S_{Football}$ is an end-vertex. Furthermore, we distinguish between two types of intermediate results: *local* and *remote*. For any vertex type, local intermediate results are intermediate data *generated* locally, while remote intermediate results are intermediate data *located* locally after receiving corresponding auxiliary data from neighboring vertices.

In principle, we define: (1) the time spent at any vertex, V, to generate local results as T^l_V, (2) the time to transmit remote auxiliary data from a sending vertex, $V1$, to a receiving vertex, $V2$, as T^r_{V1-V2}, and (3) the time to join local and remote intermediate data at a mid-vertex or an end-vertex, $V1$, as T^j_{V1-V2}, assuming remote auxiliary data are sent by vertex, $V2$[10]. We note that start-vertices do not join local and remote intermediate data because they do not receive remote auxiliary data (i.e., they have no incoming edges). In addition, although all types of vertices are initially run in parallel, a mid-vertex or an end-vertex cannot start joining local and remote intermediate results before finishing local data generation and receiving remote auxiliary data. Consequently, any mid-vertex or end-vertex, $V1$, with a neighboring sender vertex, $V2$, cannot join intermediate results before time T^{wait}_{V1-V2} = max$\{(T^l_{V2} + T^r_{V2-V1})$, $T^l_{V1}\}$. Hence, the total time needed by $V1$ to complete joining intermediate data becomes $T^{total}_{V1-V2} = T^{wait}_{V1-V2} + T^j_{V1-V2}$.

Given the above definitions, we exemplify how the query planner estimates the cost (in time) required to satisfy any generated graph plan. To streamline discussion, we consider again the graph plan G'_{P1} in Fig. 8 (b), which incorporates one start-vertex, one mid-vertex, and one end-vertex. Graph plans with multiple mid-vertices, but single start-vertex and single end-vertex are common. On the other hand, graph plans with multiple start-vertices, mid-vertices and end-vertices are still possible (e.g., we observed some in the YAGO2 benchmark suite [18, 7]) and our query planner handles them all. For simplicity, we further denote the three vertices $S_{?Country} = \{g_1, g_2\}$, $S_{?Tournament} = \{g_5, g_3\}$ and $S_{Football} = \{g_2, g_4\}$ in Fig. 8 (b) as Country, Tournament and Football, respectively.

Initially, DREAM triggers the three vertices, Country, Tournament and Football in Fig. 8 (b) concurrently. After Country is done, it sends its intermediate auxiliary data to Tournament and Football. Subsequently, both vertices, Tournament and Football, execute in parallel and join intermediate results. Clearly, Tournament and Football will complete this step after times $T^{total}_{Tournament-Country}$ and $T^{total}_{Football-Country}$, respectively. Lastly, Tournament sends its intermediate auxiliary data to Football, which, in turn, joins their corresponding triple data with its local intermediate data and emits the final result. Of course, Football cannot apply a concluding join and produce the query result before finishing its intermediate data generation and receiving intermediate auxiliary data from Tournament. Therefore, the time needed by Football to generate the final result, which is essentially composed by the query planner as an Estimated Time Equation (ETE), becomes:

$$ETE = \max\{T^{total}_{Football-Country}, (T^{total}_{Tournament-Country} + T^r_{Tournament-Football})\} + T^j_{Football-Tournament} \quad (1)$$

[10] As mentioned earlier, slave machines exchange only triple ids. The time to locate actual triples using triple ids is part of the join time.

The time spent at any slave machine, M, to generate local results is a measure of cost and is estimated as a function of the size of triples visited in M's RDF store. In contrast, the time spent by M to transmit intermediate auxiliary data is a measure of selectivity and is estimated as a function of the size of auxiliary data shuffled. To join intermediate results, we use a hash-based join algorithm and, akin to relational query optimizers, add a disk cost of 3 $\times(K+L)$ per a join operation, where K and L are the numbers of pages encompassing local and remote triples, respectively [33].

Collecting statistics for the estimated numbers of triples (or pages) visited and generated per a graph plan, G'_P, is a straightforward process in DREAM. In particular, our query planner simply relies on RDF-3X [30], which is employed at each slave machine. To elaborate, the query planner sends each vertex in G'_P to a different slave machine, M, alongside a statistics collection request. RDF-3X at M collects statistics for the received G'_P's vertex via using either conventional join estimation techniques or mining frequent join paths (see [30] for more details). Afterwards, M filters the collected statistics and sends only the estimated numbers of visited and generated triples to the master node. The query planner at the master node receives all the estimated numbers from slave machines, converts them to numbers of pages of data and auxiliary data[11], and evaluates the Estimated Time Equation, ETE (see Equation (1)), of G'_P accordingly.

3.2.5 Adaptive DREAM

The lowest-cost plan G'_P generated by the query planner can encompass more than one join vertex. As discussed in Section 3.2.3, the number of join vertices in G'_P dictates the number of slave machines needed to execute G'_P. Some SPARQL queries, however, might not need a distributed system with a number of machines amounting to the number of join vertices. Moreover, some SPARQL queries are even very simple and might not necessitate a distributed system whatsoever (i.e., a solo machine suffices). In principle, what should dictate the number of machines for G'_P are the system resources (mainly memory) needed by G'_P, and not its number of join vertices. Hence, to effectively satisfy G'_P, we suggest examining the prospect of having a number of machines less than G'_P's number of join vertices, assuming a number of join vertices greater than one (if G'_P's number of join vertices equals to 1, a single machine will be utilized directly). For instance, if G'_P has 5 join vertices, we check whether using 1, 2, 3, or 4 machines is better than using 5 machines for satisfying G'_P. That is, we always explore the full continuum of potential numbers of machines, N, where $1 \leq N \leq$ maximum number of join vertices in G'_P, and select the number of machines that will expectedly result in the best performance.

We materialize our proposal via *compacting* the lowest-cost plan G'_P of a query graph G gradually, all the way until getting a single join vertex. Specifically, if the number of join vertices in G'_P is greater than one, we treat G'_P as a new query graph and input it to the query planner. Subsequently, the query planner compacts G'_P (i.e., merges two of its join vertices) and generates a new respective graph plan, G''_P, which exhibits the minimum cost[12]. Again, if the number of join vertices in G''_P is still greater than one, G''_P is input to the query planner, which, in turn, compacts it and generates a pertaining lowest-cost plan G'''_P. The process continues until a graph plan with only a single join vertex is obtained. Finally, the graph plan with the minimum estimated cost is selected (from among all the generated lowest-cost plans) and executed. This way, DREAM adaptively elects either centralized or distributed system (with potentially different numbers of machines for different queries), depending on which system would essentially suit better the given SPARQL query graph, G.

Table 2: Our employed datasets.

Dataset	Number of Triples	Size
LUBM 20K	1 X 10^9	450 GB
LUBM 30K	3 X 10^9	700 GB
LUBM 40K	5 X 10^9	950 GB
LUBM 50K	7 X 10^9	1.2 TB
YAGO2	320 X 10^6	50 GB

4. EXPERIMENTS

4.1 Methodology

We fully implemented DREAM[13] using C and MPICH 3.0.4[14]. As an RDF store per each slave machine we utilized RDF-3X 0.3.7[15]. In this section, we thoroughly evaluate DREAM and compare it against three closely related systems, namely RDF-3X [30], Huang et *al.* [25] and H2RDF+ [32]. We refer, henceforth, to the system proposed by Huang et *al.* as GP (which stands for ***G***raph ***P***artitioning, employed by the system). RDF-3X is a centralized scheme and lies under Quadrant-I, while GP and H2RDF+ are distributed systems and lie under Quadrants II and III, and Quadrant-II, respectively. GP applies replication and leverages RDF-3X, while H2RDF+ shares the adaptivity objective with DREAM. Sections 2 and 5 describe the relevance and differences of RDF-3X, GP and H2RDF+ versus DREAM. We use the open-source codes of RDF-3X 0.3.7 and H2RDF+ 0.2[16]. We faithfully implemented and verified GP using undirected 2-hop guarantee (as described in [25]), Java 1.7u51, METIS 5.1.0 [19] and Hadoop 2.2.0 [20].

We conduct all our experiments on a private cluster and on the Amazon EC2 platform. Our private physical cluster is composed of 10 Dell PowerEdge rack-mounted servers with identical hardware, software and network capabilities. Each server incorporates 2×3.47 GHz X5690 6-core Xeon CPUs, 144 GB of RAM, 2×10 GbE, and 2×900 GB 10k rpm SAS storage (RAID 1) and runs ESXi 5.0 hypervisor. The physical cluster is managed with VMware vSphere 5.0 and a virtual cluster of 10 Virtual Machines (VMs) is provisioned to run the experiments. Each VM is located on a separate physical host and configured with 4 vCPUs, 48 GB of RAM and 320 GB of local disk[17]. The major software on each VM is 64-bit Fedora 13 Linux, MPICH 3.0.4, Apache Hadoop 2.2.0, and Oracle JDK 7u51.

We use two standard and popular benchmark suites, LUBM [22] (which is synthetic) and YAGO2 [18] (which is real). LUBM offers an ontology for academic information (e.g., universities), RDF datasets that can be generated with different sizes via controlling the number of universities, and fourteen extensional queries. YAGO2 is derived from Wikipedia, WordNet and GeoNames, and features nine queries representing a variety of properties [7]. For LUBM, we generated four datasets with 20K, 30K, 40K, and 50K universities (the latter three are used in a scalability study), resulting in 1, 3, 5 and

[11] The query planner assumes T time to read a page and $\alpha \times T$ time to shuffle a page. The α parameter is usually assigned a value >1, because communication time is typically larger than disk time in most systems. Like traditional relational query optimizers, we only focus on disk and network times, and ignore computation time.

[12] We note here that a single join vertex can be merged with *one* or *many* directly connected join vertices.

[13] All DREAM material, including code can be found at: http://www.qatar.cmu.edu/~mhhammou/DREAM/.

[14] http://www.mpich.org/

[15] https://code.google.com/p/rdf3x/

[16] https://code.google.com/p/h2rdf/

[17] Note that this disk size was enough to fit a 450 GB dataset (see Table 2) using RDF-3X due to the exhaustive compression that RDF-3X applies internally.

Table 3: Our adopted SPARQL queries.

LUBM	# of Join Vertices	YAGO2	# of Join Vertices
L2	3	A1	4
L4	1	A2	5
L7	2	B1	5
L8	2	B2	3
L9	3	C2	3
L12	2	N/A	N/A

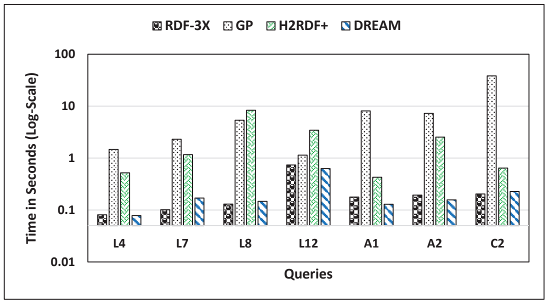


Figure 9: Runtime results for fast queries.

7 billions of triples, or 450, 700, 950 and 1.2× 10^3 GB of data, respectively. We also tested the benchmarked systems with a representative set of queries from LUBM, namely 2, 4, 7, 8, 9 and 12, which we denote as L2, L4, L7, L8, L9 and L12, respectively (the rest of the queries exhibit similar characteristics, thus were discarded). For YAGO2, we extracted 320 millions of triples, or 50 GB. Likewise, we selected A1, A2, B1, B2, and C2 as a representative mixture of queries. Tables 2 and 3 summarize our employed datasets and query workloads.

4.2 DREAM Versus Related RDF Systems

We start by discussing performance results in Figures 9 and 10. First, we classify LUBM and YAGO2 queries into two categories, *fast* and *slow* queries. Queries that finish in less than 1 second on RDF-3X are categorized as fast, while the rest are classified as slow. Based on this categorization, queries $L4$, $L7$, $L8$, $L12$, $A1$, $A2$ and $C2$ are classified as fast, while queries $L2$, $L9$, $B1$ and $B2$ are classified as slow. Fast queries are fast on RDF-3X because they usually use few index lookups and avoid full scans of the RDF dataset, thus, circumventing memory thrashing. For this type of queries, a centralized scheme like RDF-3X is generally expected to perform well. On the other hand, a distributed system can still perform well, but only if the gain from distributed execution and aggregate memories is not offset by the loss from communication and other lateral processing units (e.g., extra joins, Hadoop and/or HBase).

Table 4 shows that DREAM ran $A1$ and $A2$ as distributed using 4 machines for each of them. As discussed in Section 3.2.5, DREAM can judiciously estimate the costs of running a query using different settings and adaptively select the *sweet* (or lowest-cost) configuration, which minimizes communication and maximizes parallelism. As illustrated in Table 4, very small communication traffic are imposed by $A1$ and $A2$, hence, the loss from communication overhead is not expected to offset the gain from distributed execution and aggregate memories. Consequently, DREAM chose sweet graph plans with 4 join vertices for both, A1 and A2 and, subsequently, outperformed RDF-3X by 27.1% and 18.6%, respectively (see Fig. 9). On the flip side, Table 4 shows that $L4$, $L7$, $L8$, $L12$ and $C2$ were all run by DREAM as centralized. A special case is $L4$ which incorporates only a single join vertex, hence, DREAM irrespectively executes it as centralized because it can be maximally run on a solo machine (recall that each join vertex in a graph plan is mapped to a single machine- see Section 3.2.3). In contrast, $L7$, $L8$, $L9$, $L12$ and $C2$ exhibit 2, 2, 3, 2 and 3 join vertices, respectively. For these queries, DREAM selected sweet graph plans with 1 vertex. Accordingly, DREAM demonstrated comparable results to RDF-3X as depicted in Fig. 9, with an average degradation of 14.8%. We note that *DREAM does not seek to outperform RDF-3X when runs centralized, but rather seeks to run centralized using RDF-3X when a query is better suited for centralized execution*. The degradation experienced by DREAM when runs centralized is due to the *little* time spent by the query planner to generate and activate sweet graph plans. For instance, $C2$ is very short (it takes only 0.2 seconds on RDF-3X) and the query planner takes 0.024 seconds to generate its sweet graph plan, thus making the degradation somehow noticeable.

Table 4: Network traffic results (in Bytes) for the three competitor distributed systems (RDF-3X is centralized, thus does not incur data shuffling).

Query	GP	H2RDF+	# of Nodes Used By H2RDF+	DREAM	# of Nodes Used By DREAM
L2	36864	225443840	4	60	3
L4	9949184	152576	1	0	1
L7	538624	218112	1	0	1
L8	95232	643072	2	0	1
L9	1089470464	876609536	6	959447040	2
L12	9232384	407552	1	0	1
A1	351012	286720	6	60	4
A2	3241041	201728	5	60	4
B1	17127261	421888	5	147456	2
B2	9815928	582656	4	166912	3
C2	3562149	3675136	4	0	1

To the contrary of DREAM, GP performs much worse than RDF-3X for the fast queries. In particular, GP slackens $L4$, $L7$, $L8$, $L12$, $A1$, $A2$ and $C2$ by 18.1x, 22.7x, 41x, 1.5x, 45.7x, 37.5x and 187.9x (note the log-scale of Fig. 9), respectively. First, queries $L4$, $L7$, $L8$ and $L12$ expose graph pattern diameters less than 4 (i.e., do not exceed 2 hops). Hence, they all run in a Parallelizable Without Communication (PWOC) mode under GP (i.e., answered without triggering Hadoop for data shuffling)[18]. Nonetheless, any query under GP will still incur communication and computation overheads to transfer (large) partial results to the GP's master machine (see Table 4), apply a union to aggregate the results, and perform an extra join to remove duplicate triples (more precisely, to filter out sub-graph matches that are centered on a vertex that is not a base vertex of a partition- see [25] for more details). Evidently, these overheads become more pronounced with the fast queries, which are inherently short. In contrast to GP, DREAM averts all these overheads by adaptively selecting sweet configurations for all such queries and naturally (by design) precluding duplicate results. Aside from $L4$, $L7$, $L8$ and $L12$, $A1$, $A2$ and $C2$ are more than 2 undirected hops long, thus do not run in a PWOC mode under GP. As such, $A1$, $A2$ and $C2$ induce additional overhead due to triggering Hadoop for data redistribution. On average, DREAM accomplishes a speedup of 49.6x versus GP for the fast queries.

Similarly, H2RDF+ executes much slower than RDF-3X for the fast queries. Specifically, H2RDF+ degrades the performance of $L4$, $L7$, $L8$, $L12$, $A1$, $A2$ and $C2$ by 6.3x, 11.5x, 63.9x, 4.6x, 2.4x, 13.1x and 3.1x, respectively. First, $L4$, $L7$ and $L12$ contain some selective patterns and produce miniature results, hence, are executed by H2RDF+ as centralized. Nonetheless, H2RDF+ remains inferior to RDF-3X, mainly because of the high seek latency it incurs upon accessing its distributed HBase indices and shuffling pertaining data (Table 4 shows that network traffic are still transferred for $L4$, $L7$

[18] We note that for queries that run in a PWOC mode, GP can potentially run faster if the underlying cluster is scaled out further (we use a cluster of 10 VMs).

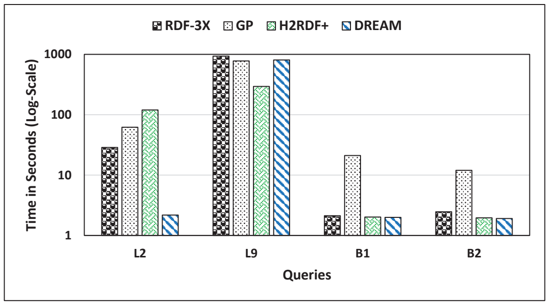


Figure 10: Runtime results for slow queries.

and $L8$ although they are run as centralized). Second, H2RDF+ executes $L8, A1, A2$ and $C2$ in a distributed fashion and, accordingly, induces network overhead (see Table 4) and still adds seek latency for accessing its distributed HBase indices. As opposed to H2RDF+, DREAM effectively adapts to query loads, avoids shuffling actual data (it reduces network traffic by an average of 99.9% versus H2RDF+ for the fast queries), and precludes the overhead of accessing distributed sophisticated indices via simply relying on RDF-3X. On average, DREAM outperforms H2RDF+ by 13.9x for the fast queries.

Aside from the fast queries, Fig. 10 portrays results for the slow LUBM and YAGO2 queries using DREAM, RDF-3X, GP and H2RDF+. For this type of queries, a distributed scheme is *generally* expected to perform better than a centralized one (this is not always the case as we will see shortly for GP and H2RDF+). As illustrated in Table 3, L2, L9, B1 and B2 involve 3, 3, 5 and 3 join vertices, out of which DREAM generated sweet graph plans with 3, 2, 2 and 3 vertices (i.e., all distributed), respectively. This resulted in 92.4% (or 13.2x), 13.4%, 6% and 22.5% runtime improvements for L2, L9, B1 and B2 against RDF-3X, respectively (note the log-scale of Fig. 10). In contrast, GP speeds up L9 by $\sim$1.2x (or 16.5%), while slacking L2, B1 and B2 by 2.1x, 9.9x and 4.8x, respectively as compared to RDF-3X. L2 is less than 2 undirected hops long (i.e., runs in PWOC mode), yet RDF-3X surpasses GP in satisfying it. This is because the loss (say, l) from communication and extra computations (i.e., a final union and an additional join) offsets the gain (say, g) from distributed execution. GP does not apply any adaptive mechanism to locate a number of machines that properly suits L2, making l more prominent, especially that L2 takes only 28.7 seconds on RDF-3X. L9, however, runs in PWOC mode, but takes 935.6 seconds on RDF-3X. This allows amortizing l and accentuating g, leading to a $\sim$1.2x speedup versus RDF-3X. To the contrary of L2 and L9, B1 and B2 are longer than 2 undirected hops (i.e., they trigger Hadoop for data redistribution), thus causing extra overhead imposed by Hadoop and, subsequently, performance degradations of 9.9x and 4.8x against RDF-3X, respectively. As opposed to GP, DREAM provides an average speedup of 11.1x for the slow queries.

Fig. 10 depicts results for H2RDF+ as well. First, H2RDF+ degrades the performance of $L2$ by 4.1x versus RDF-3X. This is due to the following three factors: (1) the high network traffic induced by $L2$ under H2RDF+ (see Table 4), (2) the overhead of accessing distributed indexes in HBase, which renders more visible when the query is not very slow (as mentioned earlier, $L2$ takes 28.7 seconds on RDF-3X) and (3) the query plan chosen by H2RDF+'s cost function for $L2$, which is sometimes far from optimal (H2RDF+ utilizes 4 machines for $L2$). On the other hand, H2RDF+ reduces the runtimes of $L9, B1$ and $B2$ by 68.4% (or 3.1x), 4.8% and 20.8%, respectively as compared to RDF-3X. The improvement of L9 is evident due to the fact that it is a very slow query (it takes 935.6 seconds on RDF-3X), allowing thereby the aforementioned overheads imposed by H2RDF+ to be amortized through long query processing times. Indeed, H2RDF+ excels with very long, non-selective queries as discussed in [32]. As opposed to H2RDF+, DREAM achieves an average speedup of 13.6x for the slow queries.

Table 5: Data versus auxiliary data (in Bytes) shuffled under DREAM around the cluster network.

Query	Data	Auxiliary Data	% of Saving in Network Traffic
L2	60	60	0%
L9	5959057408	959447040	83.8%
A1	60	60	0%
A2	60	60	0%
B1	372736	147456	60.43%
B2	463872	166912	64%

In summary, DREAM executes all the tested queries according to the two main goals set distinctly in its design, namely *adaptive execution* and *minimal communication*. As shown in Figures 9 and 10, DREAM performs well for the slow and fast queries, hence, renders general-purpose, while GP and H2RDF+ perform well (on average) for only the slow queries. To elaborate, this is attained via: (1) adopting the Quadrant-IV paradigm, which suits all types of queries (see Section 2), (2) successfully adapting to any query type and running it as either centralized or distributed depending on its complexity, and (3) achieving minimal communication through avoiding data shuffling altogether and shipping only auxiliary data around the cluster network. Table 4 shows that DREAM accomplishes reductions in network traffic of 16% and 13.4%, on average, versus GP and H2RDF+, respectively (RDF-3X is centralized and does not generate network traffic). These reductions can be referred to two main factors: (1) the query planner, which attempts to exploit parallelism, yet minimizes the number of machines needed to run a query (when 1 join vertex is used, minimal network traffic is induced), and (2) the shuffling of auxiliary data instead of actual triples. Table 5 illustrates the improvement in network traffic obtained by DREAM as a result of realizing the second factor (only queries that run distributed are shown). By transmitting only auxiliary data, DREAM decreases the network traffic by an average of 34.7%. For $L2, A1$ and $A2$ shown in Table 5, no difference between data and auxiliary data was observed due to the fact that our query planner selected graph plans which involve constituent sub-graphs that return empty results. As each sub-graph of a query graph is mapped to an independent machine (only 2 and 3 machines were utilized for these queries), minimal intermediate traffic (only control messages) were transmitted by the sub-graphs which expose no output results. $L2$, $A1$ and $A2$ eventually return empty result sets.

4.3 A Scalability Study

We now study how DREAM scales over a range of datasets. In particular, we tested DREAM on the Amazon EC2 platform over three other LUBM datasets with 30K, 40K, and 50K universities, which evaluate to 3, 5, and 7 billions of triples or 700 GB, 950 GB, and 1.2 TB (see Table 2), respectively. We provisioned a cluster of 5 EC2 instances, each of type **m3.xlarge**, which is pre-configured with 15 GB of RAM, 2 vCPUs, 2 × 40 GB SSD Storage and Ubuntu Server 14.04. We further attached two Elastic Block Storage (EBS) volumes to each instance, with 1 TB capacity per a volume, and applied RAID0 configuration across the two volumes. As for queries, we selected the slow queries L2 and L9 since they were executed as distributed by DREAM over the LUBM 20K dataset (see Section 4.2). Fig. 11 demonstrates the runtimes of L2 and L9. DREAM produced graph plans with 3 and 2 vertices for L2 and L9, respectively and scaled very well as clearly depicted in the figure.

We have also tested the case where RDF-3X is given a CPU capacity and a memory size that equates (or even exceeds) the aggregate

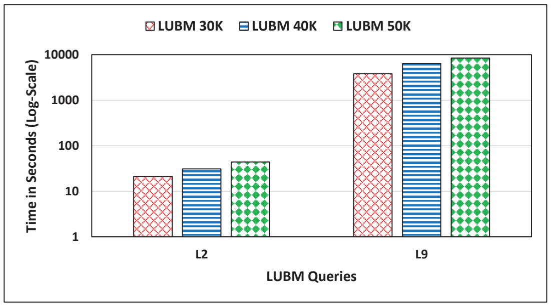


Figure 11: Runtime results of DREAM for two slow LUBM queries over various dataset sizes.

CPU and memory capacities assigned to DREAM. The objective is to verify whether a centralized RDF scheme (in this case RDF-3X) with high CPU power and large memory capacity can track or even outstrip a distributed RDF system (in this case DREAM). Evidently, with graph plans encompassing 2 and 3 vertices for L2 and L9, DREAM uses 2 and 3 machines, respectively. Since we utilized EC2 instances of type **m3.xlarge** to conduct a scalability study for DREAM, the aggregate CPU and memory capacities become 4 vCPUs and 30 GB with 2 machines, and 6 vCPUs and 45 GB with 3 machines. For a fair comparison against DREAM, we first provisioned an EC2 instance of type **m3.2xlarge** for RDF-3X. The type **m3.2xlarge** is pre-configured with 30 GB of RAM, 8 vCPUs, 2 × 80 GB SSD Storage and Ubuntu Server 14.04. Clearly, with this instance type, identical memory capacity and extra CPU power are given to RDF-3X for running L9 as opposed to DREAM with 2 **m3.xlarge** instances. As for running L2, extra CPU power, yet smaller memory size are provided to RDF-3X with this instance type versus DREAM (more memory capacity is given to RDF-3X for running L2 shortly).

Table 6: Performance of RDF-3X for L2 and L9 over a range of LUBM datasets using different EC2 instances.

Query	LUBM Dataset	EC2 Instance Type	Runtime (In Seconds)
L2	30K	m3.2xlarge (30 GB of RAM and 8 vCPUs)	48
	40K		FAILED
	50K		FAILED
L9	30K	m3.2xlarge	FAILED
	40K		FAILED
	50K		FAILED
L2	30K	r3.2xlarge (61 GB of RAM and 8 vCPUs)	35
	40K		89
	50K		144
L9	30K	r3.2xlarge	2760
	40K		5880
	50K		7680

Table 6 shows the results of running L2 and L9 under RDF-3X with LUBM 30K, 40K and 50K using an **m3.2xlarge** instance. A main observation is that L9 failed under RDF-3X over an **m3.2xlarge** machine, while it performed well under DREAM using 2 **m3.xlarge** instances (see Fig. 11). Likewise, L2 failed with LUBM 40K and 50K, but survived with LUBM 30K. For L2 with LUBM 30K, DREAM outpaced RDF-3X by 22.3x. The results clearly indicate the advantage of a distributed scheme over a centralized one. We attribute this advantage to the way the query planner of DREAM partitions the query graphs, $G1$ and $G2$, of L2 and L9, respectively. Specifically, the set of partitioned sub-graphs of $G1$ or $G2$, with each sub-graph getting eventually mapped to a different machine, results in less numbers of triples visited and generated at each machine (i.e., a smaller working set at each machine) versus running $G1$ or $G2$ as a whole on a solo machine (as is the case with RDF-3X). In fact, RDF-3X is known to perform inferiorly (and sometimes fail- see [32]) when the underlying single main memory is greatly dwarfed by the size of a query working set. Contrarily, if a distributed RDF system can better (and not necessarily fully) hold the working set of a query within its aggregate memories, as well as offset the imposed network traffic, it can highly expedite the query performance. This is the case of L2 and L9 under DREAM as related to RDF-3X.

Second, we provisioned an EC2 instance of type **r3.2xlarge** to provide more memory to RDF-3X, especially for running L2, which under DREAM it used 3 **m3.xlarge** instances (i.e., 45 GB of aggregate memory). The type **r3.2xlarge** is pre-configured with 61 GB RAM, 8 vCPUs, 2 x 800 GB SSD Storage and Ubuntu Server 14.04. Hence, with this instance type, for L2, RDF-3X is granted 15 GB extra memory capacity as compared to DREAM. Nevertheless, DREAM still outperforms RDF-3X by an average of 2.6x across the 3 LUBM datasets for L2 (Table 6 shows the results for RDF-3X). Again, this corroborates the efficacy of using a distributed RDF system like DREAM with an effective query planner at large-scale. Lastly, to further pursue a fair evaluation of DREAM against RDF-3X using an **r3.2xlarge** instance for L9, we provisioned 2 **m3.2xlarge** instances for DREAM to run L9, thus offering it equivalent memory size (i.e., 60 GB of memory), but still less CPU power, as compared to RDF-3X. Yet again, DREAM surpassed RDF-3X by an average of 2x across the three LUBM datasets for L9.

To this end, we note that DREAM parallelizes queries at the granularity of join vertices (i.e., each join vertex is mapped and executed fully on only a single machine). Hence, if a slow query contains only one join vertex, then DREAM will expose the limitation of a centralized scheme at large-scale[19]. Parallelizing SPARQL queries at different granularities (e.g., at fine-grained regular vertex, hence, partitioning a join vertex itself, and at coarse-grained join vertex) is beyond the scope of this work, yet is a promising direction and can potentially accelerate DREAM further. We set this as a main future direction for DREAM's next version (see Section 6).

4.4 DREAM with Batch Processing

As pointed out in Sections 3.2 and 4.2, the maximum number of machines that can be assigned to any query, Q, in DREAM is bound by the number of join vertices in Q. While this allows Q to effectively utilize what it only needs from cluster resources, some slave machines might be left idle. For example, if Q incorporates 5 join vertices, yet is executed by DREAM on only 2 machines over a cluster with 5 machines, 3 machines will remain idle. To account for this underutilization, we added a unique scheduling functionality to DREAM, which essentially extends the one-query-at-a-time processing paradigm adopted by most existing distributed RDF systems. In particular, many of the current distributed RDF systems run queries in a sequential order, thus disallowing cluster *space sharing*, wherein more than one query can co-execute on the same cluster at the same time.

To allow space sharing and prevent cluster underutilization, we developed two simple job schedulers in DREAM, namely, *random* and *greedy* schedulers. To elaborate, if the master receives a *job* of queries (a job can consist of 1 or many queries), it triggers the query planner, which, in turn, generates the lowest-cost graph plans of all the queries in the job and stores them in a queue. Afterwards, either the random or the greedy scheduler is enabled. The random job scheduler selects randomly as many graph plans as the cluster can fit (one after the other) from the queue and executes them *concurrently* over the cluster. If at any time, a graph plan, G_P, with a number of vertices greater than the (so far) available number of machines is

[19] As pointed out in Section 4.2, in contrast to slow queries, fast queries use few index lookups and avoid full scans of RDF datasets, thus, typically preclude memory thrashing. As such, even if a fast query contains only one join vertex, it is not expected to perform inferiorly or fail at large-scale.

chosen, the scheduler retains G_P in the queue and randomly picks another potential graph plan. This process continues until the queue becomes empty. In contrast, the greedy job scheduler selects graph plans (one after the other) in an ascending order based on their numbers of vertices, and executes as many of them as the cluster can fit at a time (we assume that the cluster can always fully fit the graph plan with the largest number of vertices). Again, the scheduler continues selecting and executing the graph plan with the smallest number of vertices from the queue, until the queue renders empty.

Of course, to pursue the suggested random and greedy job schedulers in DREAM, the master needs always to keep track of which (and indirectly how many) slave machines are available at any scheduling point in time. In fact, this can be accomplished very easily in DREAM. Specifically, each time a query is to be scheduled, the Ids of the machines that will be occupied by the query (which are decided by the query planner) are maintained in a very small *meta-graph-plans table* at the master space. In return, when a query commits, the meta-graph-plans table is updated accordingly. When a scheduling decision is to be made, the meta-graph-plans table is inspected. To this end, we tested DREAM with our random and greedy job schedulers via encompassing all the 14 standard LUBM queries as one job and running it over the LUBM 20K dataset. We observed comparable job performance (i.e., the total time to finish the whole job) improvement (on average, 63.1%) for the random and greedy schedulers versus the traditional sequential job scheduler. In the future, we plan to devise more sophisticated scheduling algorithms (see Section 6).

5. RELATED WORK

Much work has been done to effectively store and query RDF data. Indeed, it is not possible to do justice to this large body of work in this short article. Consequently, we briefly describe some of the most closely related prior centralized and distributed RDF systems.

Centralized RDF Systems: Traditional centralized systems store RDF triples in gigantic relations or hash tables such as Jena [38], Sesame [10], FORTH RDF Suite [2], rdfDB [12], 3store [15], DLDB [31], RStar [28] and Oracle [11]. Some other systems maintain RDF data in native graph forms such as the ones in [3, 17, 26, 29]. Recently, Abadi et *al.* [1] promoted a vertical partitioning approach, whereby RDF triples are grouped based on predicates and mapped onto *n* two-column tables for *n* unique predicates. This type of RDF stores is typically referred to as *predicate-oriented* stores [9]. Virtuoso 6.1.5 Open-Source Edition[20] is another example of predicate-oriented RDF systems. Hexastore [37] generalized the vertical partitioning approach via indexing RDF data in six possible ways, namely SPO, PSO, POS, OPS, OSP and SOP. BitMat [4] employed a memory-based, highly-compressed inverted index structure for storing RDF triples. RDF-3X [30] adopted exhaustive highly-compressed indices for all permutations of (S, P, O) and their binary and unary projections. Lastly, TripleBit [40] proposed a more compacted RDF store as compared to RDF-3X, with a 2D triple matrix maintained in a highly-compacted format.

A main characteristic of RDF data is the dynamicity of schemas [9]. As such, new predicates will result in new relations if a predicate-oriented approach is utilized. To avoid this problem, DB2RDF [9] suggests an *entity-oriented* approach, where columns of a relation are not pre-assigned to any predicate. Specifically, DB2RDF keeps all instances of a single predicate in the same column and stores different predicates in the same column as well. Clearly, this leads to significant space saving as compared to predicate-oriented stores.

Finally, we note that all centralized RDF systems lie under Quadrant-I (see Fig. 2). The current version of DREAM employs RDF-3X at each slave machine. However, in principle, any other centralized RDF store can be easily incorporated (this is essentially orthogonal to the DREAM's design philosophy).

Distributed RDF Systems: Researchers have also investigated distributed RDF systems [16, 34, 25, 41, 32]. A major drawback of such systems is intermediate data shuffling, which typically hinders query performance tremendously. To minimize intermediate data shuffling, Huang et *al.* [25] promote using graph partitioning by vertex instead of simple hash partitioning by subject, object or predicate. This way, vertices that are nearby in the RDF graph can be naturally collocated and mapped onto the same machine. Furthermore, they replicate triples at the boundary of each partition according to a mechanism denoted as n-hop guarantee. In particular, the n-hop guarantee strategy ensures that vertices which are *n* hops away from any vertex at any partition, *P*, are stored at *P*, even if they do not originally belong to *P*. Accordingly, queries that can be satisfied within *n* hops of any partition will result in minimal data shuffling. However, queries that exceed *n* hops of partitions will still induce huge amounts of network traffic. This traffic is exchanged using Hadoop [20] jobs. As compared to [25], DREAM precludes data partitioning altogether, avoids using Hadoop (which can slowdown queries by tens of seconds), and employs a novel query planner which effectively partitions queries (as opposed to partitioning data). Huang et *al.* [25] adopts Quadrants II and III, while DREAM employs Quadrant-IV.

Instead of modeling RDF data as triples of subjects, predicates and objects (like in [25]), Trinity.RDF [41] proposes storing RDF datasets in their native graph forms on top of Trinity [35], a distributed in-memory key-value store. Trinity.RDF replaces joins with graph explorations in order to prune the search space and avert generating useless intermediate results. In addition, it decomposes a given SPARQL query, *Q*, into a sequence of patterns so that the bindings of a current pattern can exploit the bindings of a previous pattern. As compared to DREAM, Trinity.RDF does not distribute the decomposed patterns of *Q* across machines, but sends them *all* (as one optimized sequence) to each machine. Moreover, Trinity.RDF applies data partitioning, while DREAM does not. Trinity.RDF lies under Quadrant-II.

H2RDF+ [32] proposes an indexing scheme over HBase [21] to expedite SPARQL query processing. It maintains eight indexes, each stored as a table in HBase. Besides, it partitions RDF datasets using the HBase internal partitioner, but does not partition queries. H2RDF+ shuffles intermediate data and utilizes a Hadoop-based sort-merge join algorithm to join them. Lastly, it runs simple queries over single machines and complex ones over varied numbers of distributed machines. To the contrary of DREAM, H2RDF+ lies under Quadrant-II.

Most recently, the TriAD system [13] suggested asynchronous inter-node communication and parallel/distributed join execution over a distributed, six in-memory vectors of triples, each corresponding to one SPO permutation. TriAD applies graph summarization at the master node so as to avoid searching a large part of the data graph at slave machines. The data graph is partitioned using a locality-based, horizontal hash partitioning algorithm and stored across the six distributed in-memory vectors at the slaves. As opposed to DREAM, TriAD adopts Quadrant-II. Table 7 presents a comparison between TriAD, H2RDF+ [32], Huang et *al.* [25], Trinity.RDF [41] and DREAM.

6. CONCLUSIONS

In this paper, we presented DREAM, a ***D***istributed ***R***DF ***E***ngine with ***A***daptive query planner and ***M***inimal communication. DREAM avoids partitioning RDF datasets and, contrarily, partitions SPARQL queries. As such, it follows a novel paradigm, which has not yet been explored in literature (to our knowledge). We refer to this paradigm as the Quadrant-IV paradigm. By not partitioning datasets at cluster machines, DREAM is able to satisfy queries without shuffling intermediate data whatsoever (only auxiliary data are transferred).

[20] http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/main/

Table 7: A comparison between various distributed RDF systems and DREAM.

Scheme	Quadrant(s)	Data Store	Data Model	Partitioning Strategy	Exchange Intermediate Data	Distributes Sub-Queries
Huang et *al.* [25]	II & III	RDF-3X [30] per each cluster machine	Triple-Based	Graph partitioning using METIS [19]	Yes, when a query spans more than one partition	Yes, when a query spans more than one partition
Trinity.RDF [41]	II	Distributed in-memory key-value store (based on [35])	Graph-Based	Random	Yes	No
H2RDF+ [32]	II	HBase [21]	Triple-Based	HBase internal partitioner	Yes, but compressed	No
TriAD [13]	II	Distributed in-memory triple vectors	Graph-Based & Triple-Based	Graph (using METIS [19])& horizontal hash partitioning	Yes	No
DREAM	IV	Any (current version uses RDF-3X [30] per) each slave machine)	Any	N/A	No; only auxiliary data	Yes

On the other hand, by partitioning queries, DREAM is capable of adaptively choosing a suitable number of machines, which maximizes parallelism and minimizes auxiliary data communication, for any given query. This is accomplished via employing a novel graph-based query planner with a new cost model and a rule-oriented query partitioning strategy. Experimentation results show that DREAM can effectively achieve its design goals and outperform three related popular RDF systems, RDF-3X [30], Huang et *al.* [25] and H2RDF+ [32].

Finally, we set forth three main future directions. First, DREAM currently partitions SPARQL queries at the granularity of join vertices. We envision that partitioning queries at different granularities (e.g., at coarse-grained *join* and fine-grained *regular* vertices) would accelerate DREAM further. Second, we will develop job schedulers that are more sophisticated than the current random and greedy schedulers of DREAM (e.g., something similar to the fair and capacity schedulers in Hadoop [20]). Finally, we will study the applicability of offering an *elastic* version of DREAM, where its cluster size can be expanded and contracted on the cloud based on observed query loads.

7. REFERENCES

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

[2] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs. In *WebDB*, 2001.

[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] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*. 2007.

[6] A. Badam and V. S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In *NSDI*, 2011.

[7] R. Binna, W. Gassler, E. Zangerle, D. Pacher, and G. Specht. Spiderstore: A native main memory approach for graph storage. *Grundlagen von Datenbanken*, 2011.

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

[9] 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.

[10] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*. 2002.

[11] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient sql-based rdf querying scheme. In *VLDB*, 2005.

[12] R. Guha. rdfdb: An rdf database, 2000.

[13] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In *SIGMOD*, 2014.

[14] M. Hammoud and M. F. Sakr. Distributed programming for the cloud: Models, challenges and analytics engines. 2013.

[15] S. Harris and D. N. Gibbins. 3store: Efficient bulk RDF storage. 2003.

[16] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *The Semantic Web*. 2007.

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

[18] 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 companion*, 2011.

[19] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. Metis.

[20] http://hadoop.apache.org/. Apache hadoop.

[21] http://hbase.apache.org/. Apache hbase.

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

[23] http://www.seagate.com/. Seagate enterprise capacity 3.5 hdd datasheet.

[24] http://www.w3.org/RDF/. W3C: Resource Description Framework (RDF).

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

[26] Y. H. Kim, B. G. Kim, J. Lee, and H. C. Lim. The path index for query processing on RDF and RDF Schema. In *ICACT*, 2005.

[27] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[28] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *CIKM*, 2004.

[29] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational RDF database. In *ADC*, 2005.

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

[31] Z. Pan and J. Heflin. Dldb: Extending relational databases to support semantic web queries. Technical report, DTIC Document, 2004.

[32] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *IEEE Big Data*, 2013.

[33] R. Ramakrishnan and J. Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[34] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, 2010.

[35] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[36] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[37] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.

[38] K. Wilkinson, C. Sayers, H. A. Kuno, D. Reynolds, et al. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, volume 3, 2003.

[39] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

[40] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *PVLDB*, 6(7), 2013.

[41] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *VLDB*, 2013.