Evaluating SPARQL Queries on Massive RDF Datasets

Razen Harbi* Ibrahim Abdelaziz* Panos Kalnis* Nikos Mamoulis‡
* King Abdullah University of Science and Technology
‡ University of Ioannina, Greece
{first.last}@kaust.edu.sa nikos@cs.uoi.gr

ABSTRACT

Distributed RDF systems partition data across multiple computer nodes. Partitioning is typically based on heuristics that minimize inter-node communication and it is performed in an initial, data pre-processing phase. Therefore, the resulting partitions are static and do not adapt to changes in the query workload; as a result, existing systems are unable to consistently avoid communication for queries that are not favored by the initial data partitioning. Furthermore, for very large RDF knowledge bases, the partitioning phase becomes prohibitively expensive, leading to high startup costs.

In this paper, we propose AdHash, a distributed RDF system which addresses the shortcomings of previous work. First, AdHash initially applies lightweight hash partitioning, which drastically minimizes the startup cost, while favoring the parallel processing of join patterns on subjects, without any data communication. Using a locality-aware planner, queries that cannot be processed in parallel are evaluated with minimal communication. Second, AdHash monitors the data access patterns and adapts dynamically to the query load by incrementally redistributing and replicating frequently accessed data. As a result, the communication cost for future queries is drastically reduced or even eliminated. Our experiments with synthetic and real data verify that AdHash (i) starts faster than all existing systems, (ii) processes thousands of queries before other systems become online, and (iii) gracefully adapts to the query load, being able to evaluate queries on billion-scale RDF data in subseconds. In this demonstration, audience can use a graphical interface of AdHash to verify its performance superiority compared to state-of-the-art distributed RDF systems.

1. INTRODUCTION

The RDF data model does not require a predefined schema and is a versatile way for representing information from diverse sources. Therefore, social networks, search engines and scientific databases are adopting RDF for publishing Web content.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

As the volume of RDF data continues soaring, managing, indexing and querying very large collections of RDF data becomes challenging. Early efforts focused on building efficient centralized RDF systems; like RDF-3X [9] and TripleBit [13]. However, centralized data management and search does not scale well for complex queries on web-scale RDF data. As a result, distributed RDF management systems were introduced to improve performance. Such systems scale-out by partitioning RDF data among many computer nodes and evaluating queries in a distributed fashion. A SPARQL query is decomposed into multiple subqueries that are evaluated by each node independently. Since data is distributed, the nodes may need to exchange intermediate results for finalizing the query result. Consequently, queries with large intermediate results incur high communication cost, which is detrimental to the query performance [7, 5].

Distributed RDF systems aim at minimizing the number of decomposed subqueries by partitioning the data carefully. In other words, their goal is to partition the data such that each node has all the data it needs to evaluate the entire query, without exchanging intermediate results. Thus, in a *parallel* query evaluation, each node contributes a partial result of the query; the final query result is the union of all partial results. To do so, some triples need to be replicated in multiple partitions which allows each node to answer the query without communication. Still, even sophisticated partitioning and replication cannot guarantee that arbitrarily complex SPARQL queries can be processed in parallel; thus, expensive *distributed* query evaluation, with intermediate results exchanged between nodes cannot always be avoided.

Challenges. Existing distributed RDF systems are facing two limitations. (i) *Partitioning cost:* graph partitioning is an NP-complete problem; therefore, existing systems are based on partitioning heuristics. Systems that use simple heuristics like hash partitioning [10, 14] incur excessive communication during query evaluation, as the chances that a query can be evaluated in parallel without any communication between nodes are low. On the other hand, sophisticated partitioning heuristics [5, 7, 8, 12] suffer from high preprocessing cost and sometimes high replication. More importantly, they pay the cost of partitioning the entire data regardless of the anticipated workloads. However, as shown in a recent study [11], only a small fraction of the whole graph is actually accessed by typical real query workloads. For example, a real workload consisting of more than 1,600 queries executed on DBpedia (459M triples) touches only 0.003% of the whole data. Therefore, we argue that distributed RDF systems need to leverage query workloads in

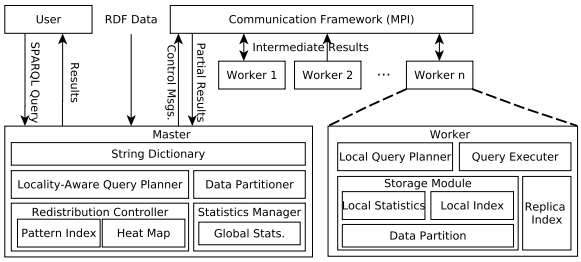

Figure 1: System architecture of AdHash

data partitioning. (*ii*) *Adaptivity:* WARP [6] and Partout [3] consider the workload during data partitioning. They achieve a significant reduction in the replication ratio, while showing better query performance compared to systems that partition the data blindly. However, both systems assume a representative (i.e., *static*) workload for data partitioning and do not adapt to changes in the workload. SPARQL query evaluation exhibits poor data locality, therefore, regardless of the partitioning heuristic used, there will always be queries that cross partition boundaries and require expensive distributed evaluation. Furthermore, because of the diversity and dynamism of workloads, another study [1] showed that systems need to continuously adapt to workloads to consistently provide good performance.

In this paper, we propose **Ad**aptive **Hash**ing (AdHash), a distributed in-memory RDF engine. It alleviates the aforementioned limitations based on the following key principles.
Lightweight Initial Partitioning: AdHash uses an initial hash partitioning, which distributes triples by hashing on their subjects. This partitioning (i) has low cost and (ii) does not incur replication. Thus, the preprocessing time is significantly reduced, partially addressing the first challenge.
Hash-based Locality Awareness: AdHash achieves competitive performance by maximizing the number of joins that can be executed in parallel without data communication by exploiting hash-based locality; the join patterns on subjects included in a query can be processed in parallel. The locality-aware query optimizer of AdHash uses this nice property to compute an evaluation plan that minimizes the size of intermediate results shipped between workers.
Adapting by Incremental Redistribution: AdHash monitors the executed workload and incrementally updates a hierarchical heat-map of accessed data patterns. Hot patterns are redistributed and potentially replicated in the system in a way that future queries that include them are executed in parallel by all workers without data communication. To control replication, AdHash operates within a budget and employs an eviction policy for the redistributed patterns. This way, AdHash overcomes the limitations of static partitioning schemes and adapts dynamically to changing workloads.

2. AdHash ARCHITECTURE

AdHash employs the typical master-slave paradigm (see Figure 1) and is deployed on a shared-nothing cluster of machines which communicate through message passing.

2.1 Master

The master begins by partitioning the data among workers and collecting global statistics. Then, it receives queries from users, generates execution plans, coordinates workers, collects final results, and returns results to users.
String Dictionary: RDF data contains long strings in the form of URIs and literals. To avoid the storage, processing, and communication overheads, we encode RDF strings into numerical IDs and build a bi-directional dictionary.
Data Partitioner: A recent study [4] showed that joins on the subject column account for 60% of the joins in a real workload of SPARQL queries. Therefore, AdHash uses lightweight hash-based triple sharding on subject values.
Statistics Manager: maintains statistics about the RDF graph, which are used for global query planning and during adaptivity. Statistics are collected in a distributed manner during bootstrapping.
Redistribution Controller: monitors the workload in the form of heat maps and triggers the adaptive Incremental ReDistribution (IRD) process for hot patterns. Only data accessed by hot patterns are redistributed and potentially replicated among workers. A redistributed hot pattern can be answered by all workers in parallel without communication. Using hierarchical representation, replicated hot patterns are indexed in a structure called Pattern Index (PI). Patterns in the PI can be combined for evaluating future queries without communication. Further, the controller implements replica replacement policy to keep replication within a threshold.
Locality-Aware Query Planner: uses the global statistics and the pattern index from the redistribution controller to decide if a query, in whole or partially, can be processed without communication. Queries that can be fully answered without communication are planned and executed by each worker independently. On the other hand, for queries that require communication, the planner exploits the hash-based data locality and the query structure to find a plan that minimizes communication and the number of distributed joins.
Failure Recovery. The master does not store any data but can be considered as a single-point of failure because it maintains the dictionaries, global statistics, and PI. A standard failure recovery mechanism (log-based recovery [2]) can be employed by AdHash. Assuming a stable storage, the master can recover by loading the dictionaries and global statistics because they are read-only and do not change in the system. The PI can be easily recovered by reading the query log and reconstructing the heat map.

2.2 Worker

Storage Module. Each worker w_i stores its local set of triples D_i in an in-memory data structure, which supports the following search operations: (*i*) given a predicate p, return set $\{(s,o) \mid \langle s,p,o\rangle \in D_i\}$. (*ii*) given a subject s and a predicate p, return set $\{o \mid \langle s,p,o\rangle \in D_i\}$.; and (*iii*) given an object o and a predicate p, return set $\{s \mid \langle s,p,o\rangle \in D_i\}$.

Since all the above searches require a known predicate, we primarily hash the triples in each worker by predicate. The resulting *predicate* index (simply P-index) immediately supports search by predicate. Furthermore, we use two hash maps to re-partition each bucket of triples having the same predicate, based on their subjects and objects, respectively. These two hash maps support the second and the third types of search and they are called *predicate-subject* and *predicate-object* indexes, respectively. Given that the number of unique predicates is typically small, our storage scheme avoids unnecessary repetitions of predicate values.

Note that when answering a query, if the predicate itself is a variable, then we simply iterate over all predicates. The storage module computes statistics about its local data and shares them with the master after data loading.

Replica Index. Each worker has an in-memory *replica index* that stores and indexes replicated data as a result of the adaptivity. This index initially contains no data and is updated dynamically by the IRD process.

Query Processor. Each worker has a query processor that operates in two modes: (*i*) *Distributed Mode* for queries that require communication. In this case, all workers solve the query concurrently and exchange intermediate results. (*ii*) *Parallel Mode* for queries that can be answered without communication. Each worker has all the data needed for query evaluation locally.

Local Query Planner. Queries executed in parallel mode are planned by workers autonomously. For example, star queries that join on the subject are processed in parallel because of the initial data partitioning. Moreover, queries that can be answered in parallel as a result of the adaptivity process are also planned by the local query planner.

2.3 System overview

Here we give an abstract overview of AdHash. After encoding and partitioning the data, each worker loads its triples and collects local statistics. The master node aggregates these statistics and AdHash starts answering queries. A user submits a SPARQL query Q to the master. The query planner at the master consults the redistribution controller to decide whether Q can be executed in parallel mode. The redistribution controller uses global statistics to transform Q into a hierarchical representation Q'. If Q' exists in the Pattern Index (PI) or if Q' is a star query joining on the subject column, then Q can be answered in parallel mode, otherwise, it is executed in distributed mode. If Q is executed in distributed mode, the locality-aware planner devises a global query plan. Each worker gets a copy of this plan and evaluates the query accordingly. If Q can be answered in parallel mode, the master broadcasts the query to all workers. Each worker generates its local query plan using local statistics and executes Q without communication. As more queries get submitted to the system, the redistribution controller updates the heat map, identifies hot patterns, and triggers the IRD process. Consequently, AdHash adapts to the query load by answering more queries in parallel mode.

3. DEMONSTRATION DETAILS

We provide an interactive GUI to demonstrate the effectiveness of AdHash using large-scale real and synthetic RDF datasets. In addition to AdHash, we also deploy other distributed RDF systems so participants can evaluate the performance of any of these systems against AdHash.

3.1 Demonstration Setup

We implemented AdHash in C++ and used MPI for synchronization and communication. Our non-adaptive version of AdHash is coined as AdHash-NA. We deploy AdHash, SHAPE, SHARD and H2RDF+ on a cluster of 12 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs. The machines run 64-bit 3.2.0-38 Linux Kernel and are connected by a 10Gbps Ethernet switch.

Datasets Specifications: Table 1 shows the characteristics of the datasets. Bio2RDF is a community project that provides linked data for life sciences. We use Bio2RDF[1] release 2 which contains 4.64 billion triples connecting 24 different biological datasets. We also use the synthetic LUBM-10240[2] dataset which contains 1.36 billion triples. WatDiv[3] dataset contains 109 million triples and provides a wide spectrum of queries with varying characteristics.

Table 1: Datasets Statistics in millions (M)

Dataset	Triples (M)	#S (M)	#O (M)	#P
Bio2RDF	4,644.44	552.08	1,075.58	1,714
LUBM-10240	1,366.71	222.21	165.29	18
WatDiv	109.23	5.21	17.93	85

Table 2: Preprocessing time (minutes)

	LUBM-10240	WatDiv	Bio2RDF
AdHash	**14**	**1.2**	**115**
METIS	523	66	4,532
SHAPE	263	79	>24h
SHARD	72	9	143
H2RDF+	152	9	387

3.2 Demonstration Interface

Figure 2 show our GUI interface for the deployed systems and datasets. We offer three different interaction scenarios:

Dataset Loading: We will demonstrate the data preprocessing time of AdHash. Participants can select a dataset to load and index in AdHash and monitor the progress of the loading phase. The loading and indexing cost of the other engines is significantly high (see Section 3.3.1). Therefore, we limit this feature to AdHash only since it is the only system that provides practical startup costs. Note that for other systems, the data are already preprocessed and indexed offline.

Query Evaluation: Participants can write any SPARQL query or choose one of the predefined queries. The predefined queries are either selected from a real query log (Bio2RDF), or they are benchmark queries (LUBM and WatDiv). Predefined queries are selected such that they provide a mixture of queries with varying structural characteristics and selectivity. Once the query and the associated dataset are defined, participants can choose the underlying execution engine from AdHash, SHAPE, H2RDF+ or SHARD. Then, the query is submitted through the GUI and evaluated using the chosen system.

Direct Comparison: By deploying different distributed engines as well as AdHash, we allow participants to compare these systems head-to-head. Participants can experience these metrics themselves using the demonstration GUI and reproduce/validate our findings.

3.3 Experimental Evaluation

3.3.1 Preprocessing Time

This experiment measures the time it takes a system for preparing the data prior to answering queries. The results are shown in Table 2 for all datasets. For TriAD, we show only the time to partition the graph using METIS[4]. As Table 2 shows, sophisticated partitioning techniques employed by TriAD and SHAPE are prohibitively expensive compared

[1]http://download.bio2rdf.org/release/2/
[2]http://swat.cse.lehigh.edu/projects/lubm/
[3]http://db.uwaterloo.ca/watdiv/
[4]http://www.cs.umn.edu/∼metis

Table 3: Query runtime (seconds)

System	LUBM-10240							WatDiv (GeoMean)				Bio2RDF					
	L1	L2	L3	L4	L5	L6	L7	L1-L5	S1-S7	F1-F5	C1-C3	B1	B2	B3	B4	B5	B6
AdHash	**0.317**	**0.120**	**0.006**	**0.001**	**0.001**	0.004	**0.220**	**0.002**	**0.001**	**0.010**	**0.012**	**0.004**	**0.002**	**0.002**	**0.004**	**0.002**	**0.001**
AdHash-NA	2.743	0.120	0.320	0.001	0.001	0.04	3.203	0.009	0.006	0.235	0.123	0.0198	0.016	0.036	0.227	0.187	0.001
Triad-SG[5]	2.15	2.02	1.65	0.001	0.001	**0.001**	16.86	0.002	0.003	0.029	0.270						
Trinity.RDF[5]	7.00	3.50	6.00	0.004	0.003	0.001	27.50										
SHAPE	25.32	4.38	25.36	1.60	1.57	1.56	15.02	1.87	1.82	1.84	2.72	NA	NA	NA	NA	NA	NA
H2RDF+	285.43	71.72	264.78	24.12	4.76	22.91	180.32	5.44	8.68	18.46	65.79	5.58	12.71	322.30	29.55	7.96	4.28
SHARD	413.72	187.31	ABORT	358.20	116.62	209.80	469.34	ABORT	ABORT	ABORT	ABORT	239.35	309.44	512.85	788.02	787.10	112.28

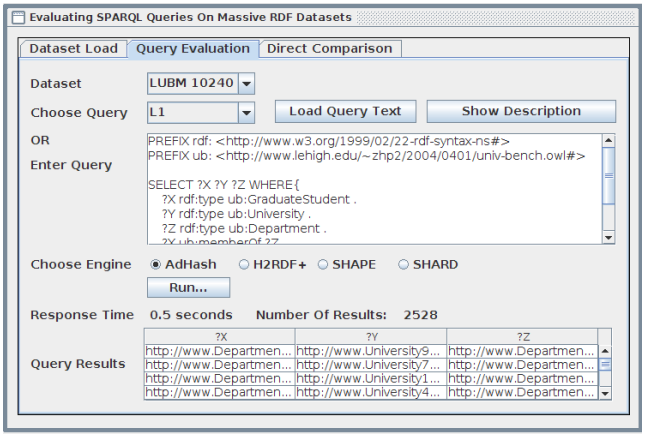

Figure 2: Demonstration Interface

to lightweight partitioning techniques adopted by SHARD and H2RDF+. For Bio2RDF, SHAPE took more than 24 hours in the partitioning phase only. Moreover, it resulted in a severe imbalance by placing 98.2% of the data in one machine while distributing the rest 1.8% among the other 11 machines. AdHash uses lightweight hash partitioning and avoids the upfront cost of sophisticated partitioning schemes. As Table 2 shows, AdHash starts 4X up orders of magnitude faster than existing systems.

3.3.2 *Query Performance*

Table 3 shows the query response times of all systems. Bio2RDF queries (B1-B6) are extracted from a real query log while we use queries L1-L7 (also used in [14, 5]) for the LUBM dataset. WatDiv queries have four categories: linear (L), star (S), snowflake-shaped (F) and complex queries (C). Queries can be classified based on their selectivity and structural complexity into two categories: simple and complex. Simple queries are L2, L4, L5, L6, S1-S7, L1-L5, B1, B3, B4 and B7 whereas complex queries are L1, L3, L7, F1-F5, C1-C3, B2, B5 and B6. SHARD and H2RDF+ suffer from the expensive overhead of MapReduce; therefore, their performance is significantly worse than all other systems for complex queries. However, H2RDF+ performs better than SHARD for simple queries as these queries are solved in a centralized fashion using HBase scanners. On the other hand, SHAPE incurs minimal communication and performs better than SHARD and H2RDF+ due to the utilization of semantic hash partitioning. Nonetheless, as it uses MapReduce for dispatching queries to workers, it still suffers from the overhead of MapReduce. In-memory RDF engines, Trinity.RDF and TriAD, perform significantly better than MapReduce-based systems. Due to the sophisticated partitioning scheme (i.e., METIS) employed by TriAD, its performance is better than Trinity.RDF, which relies on the fast underlying network. Observe that, even with simple hash partitioning, AdHash-NA achieves better or comparable performance to TriAD for all queries. Furthermore, AdHash is significantly faster than all systems and provides sub-second execution times for all query complexities.

[5]Trinity [14] and TriAD [5] are not publicly available, therefore, we compare to the numbers they reported using the same datasets they use on a comparable hardware.

4. REFERENCES

[1] G. Aluç, M. T. Özsu, and K. Daudjee. Workload Matters: Why RDF Databases Need a New Design. *PVLDB*, 7(10), 2014.

[2] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3), 2002.

[3] L. Galarraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. *CoRR*, abs/1212.5636, 2012.

[4] M. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD*, 2011.

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

[6] K. Hose and R. Schenkel. WARP: Workload-aware replication and partitioning for RDF. In *ICDEW*, 2013.

[7] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[8] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14), 2013.

[9] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.

[10] 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.

[11] L. Rietveld, R. Hoekstra, S. Schlobach, and C. Guéret. Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling. In *ISWC*, 2014.

[12] Wu, Buwen and Zhou, Yongluan and Yuan, Pingpeng and Liu, Ling and Jin, Hai. Scalable SPARQL Querying using Path Partitioning. In *ICDE*, 2015.

[13] 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):517–528, 2013.

[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.