Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration

Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems,
Shanghai Jiao Tong University

Feifei Li
School of Computing,
University of Utah

Abstract

Many knowledge bases like Google and Facebook's knowledge/social graphs are represented and stored as RDF graphs, where users can issue structured queries on such graphs using SPARQL. With massive queries over large and constantly growing RDF data, it is imperative that an RDF graph store should provide low latency and high throughput for concurrent query processing. However, prior systems still experience high per-query latency over large datasets and most prior designs have poor resource utilization such that each query is processed in sequence.

We present Wukong[1], a distributed graph-based RDF store that leverages RDMA-based graph exploration to support highly concurrent and low-latency queries over large data. Following a graph-centric design, Wukong builds indexes by extending the graphs with index vertices and leverages differentiated graph partitioning to retain locality (for normal vertices) while exploiting parallelism (for index vertices). To explore the low-latency feature of RDMA, Wukong leverages a new RDMA-friendly, predicate-based key/value store to store the partitioned graphs. To provide low latency and high parallelism, Wukong decomposes a query into sub-queries, each of which may be distributed over and handled by a set of machines in parallel. For each sub-query, Wukong leverages RDMA to provide communication-aware optimizations to balance between execution and data migration. To reduce inter-query interference, Wukong leverages a worker-obliger work stealing mechanism to oblige queries in straggler workers. Evaluation on a 6-node RDMA-capable cluster shows that Wukong significantly outperforms state-of-the-art systems like TriAD and Trinity.RDF for both latency and throughput, usually at the scale of orders of magnitude.

1 INTRODUCTION

Many large datatsets, especially for a knowledge base, are increasingly published using the Resource Description Framework (RDF) format, which represents a dataset as a set of $\langle subject, predicate, object\rangle$ triples that form a directed and labeled graph. Examples include Google's knowledge graph [11] and Facebook's social graph [31], and a number of public knowledge bases including DBpedia [1], Probase [35], PubChemRDF [18] and Bio2RDF [6]. There are also a number of public and commercial websites like Google and Bing providing online queries through SPARQL[2] to such datasets.

With the increasing scale of RDF datasets and the growing number of queries per second received by these applications, it is imperative that an RDF store provides low latency and high throughput over highly concurrent queries. In response, much recent research has been devoted to develop scalable and high performance systems to index RDF data and to process SPARQL queries. Early RDF stores like RDF-3X [19, 20], SW-Store [4], HexaStore [33] usually use a centralized design, while later designs such as TriAD [12], Trinity.RDF [39], H_2RDF [23, 22] and SHARD [25] explore a distributed store in response to the growing data sizes.

An RDF dataset is essentially a highly connected, directed graph. Hence, an RDF store may either store a set of triples as records in a relational table (i.e., a triple store) [19, 20, 12, 23, 38], or manage them as a native graph (i.e., a graph store) [38, 5, 39]. Prior work [39] shows that while using a triple store may enjoy query optimizations designed for database queries, handling SPARQL queries extensively relies on join operations over potentially large tables, which usually generates huge redundant intermediate data. Besides, using a relational store may limit the types of queries the stores can support natively, such as general graph queries like reachability analysis and community detection.

In this paper, we describe Wukong, a distributed in-memory RDF store that provides low-latency, concurrent queries over large RDF datasets. To make it easy to scale out, Wukong follows a graph-based design by storing RDF triples as a native graph and leverages graph exploration to handle queries. Unlike prior graph-based RDF stores that are only designed to handle one query at a time, Wukong is also designed to provide high throughput such that it can handle hundreds of thousands of con-

[1] Short for Sun Wukong, who is known as the Monkey King and is a main character in the Chinese classical novel "Journey to the West". Since Wukong is known for his swift reactions to complex situations and ability to do massive multi-tasking, both in large scale input (e.g. space and time), we term our system as Wukong. The source code of Wukong is available from http://ipads.se.sjtu.edu.cn/projects/wukong.

[2] An acronym for both SPARQL Protocol and RDF Query Language.

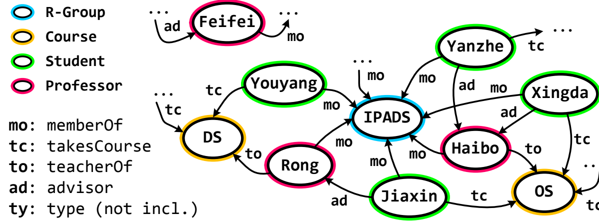

Figure 1: An example RDF graph.

current queries per second. A key design of Wukong is to use fast network primitives like one-sided RDMA as much as possible. Besides, Wukong also contributes a set of new designs and techniques for query processing over RDF graphs.

Flexible graph-based model and storage (§4). Besides storing RDF triples as a graph by treating *object/subject* as vertices and *predicate* as edges, Wukong extends an RDF graph by introducing index vertices so that indexes are naturally part of the graph. To partition and distribute data to multiple machines, Wukong applies a differentiated partition scheme [8] to embrace both locality (for normal vertices) and parallelism (for index vertices) during query processing. Besides, Wukong incorporates predicate-based finer-grained partitioning into a refined, RDMA-friendly distributed hashtable [32] for efficiently storing the RDF graph.

Fast and scalable query processing (§5). Depending on the selectivity and complexity of queries, the query execution time may vary significantly (more than 5,000X for queries in standard benchmark like LUBM [2]). Wukong decomposes a query into a sequence of subqueries and handles multiple independent subqueries simultaneously. For each subquery, Wukong adopts a RDMA communication-aware mechanism: for small (selective) queries, it uses in-place execution that leverages one-sided RDMA READ to fetch necessary data so that there is no need to move intermediate data; for large (non-selective) queries, it uses one-sided RDMA write to distribute the query processing to all related machines. Moreover, being aware of the cost-insensitivity of RDMA operations with respect to data size, Wukong leverages *full-history pruning* such that Wukong can precisely prune unnecessary immediate data and thus avoid the costly final aggregation of results from multiple machines. To avoid large queries from blocking small queries for handling concurrent queries, Wukong leverages a latency-centric work stealing scheme to dynamically oblige queries in straggling workers.

We have implemented Wukong and evaluated it on a 6-node cluster using a set of common RDF query benchmarks over a set of synthetic (e.g., LUBM and WSDTS) and real-life (e.g., DBPSB and YAGO2) datasets. Our experiment shows that Wukong provides orders of magnitude lower latency compared to centralized (e.g., RDF-3X and BitMat) and distributed (e.g., TriAD and Trinity.RDF) state-of-the-art systems. An evaluation using a mixture of LUBM queries shows that Wukong can achieve up to 185K queries per second on 6 machines with 0.80 milliseconds (geometric mean) median latency.

2 Background

2.1 RDF and SPARQL

The RDF dataset is a graph (aka RDF graph) composed by triples, where a triple is formed as $\langle subject, predicate, object \rangle$. A triple can be regarded as a directed edge (*predicate*) connecting two vertices (from *subject* to *object*). Thus, an RDF graph can be alternatively viewed as a directed graph $G = (V, E)$, where V is the collection of all vertices (subjects and objects), and E is the collection of all edges, which are categorized by their labels (predicates). W3C has provided a set of unified vocabularies (as part of the RDF standard) to encode the rich semantics, where the rdfs:type predicate (or type for short) provides a classification of vertices of an RDF graph into different groups. As shown in Figure 1, a simplified sample RDF graph of LUBM dataset [2], the entity Feifei has type Professor[3], and there are four categories of edges linking entities, namely, memberOf (mo), takesCourse (tc), teacherOf (to), and advisor (ad).

SPARQL, a W3C recommendation, is the standard query language for RDF datasets. The most common type of SPARQL queries is as follows:

Q := SELECT RD WHERE GP

where, GP is a set of *triple patterns* and RD is a *result description*. Each triple pattern is of the form $\langle subject, predicate, object \rangle$, where each of the subject, predicate and object may denote either a *variable* or a *constant*. Given an RDF data graph G, the triple pattern GP searches on G for a set of subgraphs of G, each of which matches the graph pattern defined by GP (by binding pattern variables to values in the subgraph). The result description RD contains a subset of variables in the graph patterns.

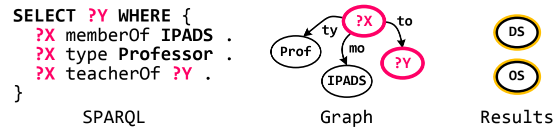

Figure 2: A SPARQL query (Q_1) on sample RDF graph.

For example, as shown in Figure 2, the query Q_1 retrieves all objects that were taught (to) by a Professor who is a member (mo) of IPADS. The query can also be graphically represented by a query graph, in which vertices represent the subjects and objects of triple patterns; black vertices represent constants, and red vertices represent variables; Edges represent predicates in the required

[3]To save space, we use color circles to represent the type of entities.

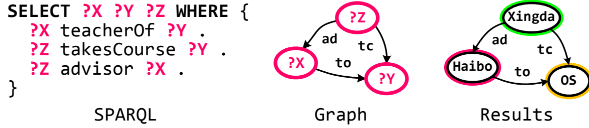

Figure 3: A SPARQL query (Q_2) on sample RDF graph.

patterns (GP). The query results (?Y, described in RD) include DS and OS.

2.2 Existing Solutions

We next discuss two representative approaches adopted in existing state-of-the-art RDF systems.

Triple Store and triple Join: A majority of existing systems store and index RDF data as a set of *triples* in relational databases, and excessively leverage triple *join* operations to process SPARQL queries. Generally, query processing consists of two phases: Scan and Join. In the Scan phase, the RDF engine decomposes a SPARQL query into a set of triple patterns. For the query in Figure 2, the triple patterns are {?X memberOf IPADS}, {?X type Professor} and {X? teacherOf ?Y}. For each triple pattern, it generates a temporary query table with bindings by scanning triple store. In Join phase, the query tables are joined to produce the final query results.

Some prior work [39] has summarized inherent limitations of triple-store based approach. First, triple stores rely excessively on costly join operations, especially for distributed merge/hash-join. Second, the scan-join approach may generate large redundant intermediate results. Finally, while using redundant six primary SPO[4] *permutation indexes* [33] can accelerate scan operations, such indexes lead to heavy memory pressure.

Graph Store and graph exploration: Instead of joining query tables, Trinity.RDF [33] stores RDF data in a native *graph* model on top of a distributed in-memory key/value store, and leverages fast *graph-exploration* strategy for query processing. It further adopts one-step pruning (i.e., the constraint in the immediately prior step) to reduce the intermediate results. As an example, considering Q_1 in Figure 2 over the data in Figure 1, after exploring the type of Professor for each member of IPADS wrt the data in Figure 1, we find that the possible binding for ?X is only Rong and Haibo, and the rest of members are pruned.

However, the graph exploration in Trinity.RDF relies on a final centralized join to filter out non-matching results, which is a potential bottleneck [12, 22], especially for queries with cycles and/or large intermediate results. For example, the query Q_2 in Figure 3 asks for advisors (?X), courses (?Y) and students (?Z) such that the advisor advises (ad) the student who also takes a course (tc) taught by (tc) the advisor. After exploring all three triple patterns in Q_2 wrt to the data in Figure 1, the non-matching bindings, namely, Haibo to OS, OS tc Jiaxin and Jiaxin ad Rong will not be pruned until a final join.

[4] S, P and O stand for subject, predict and object accordingly.

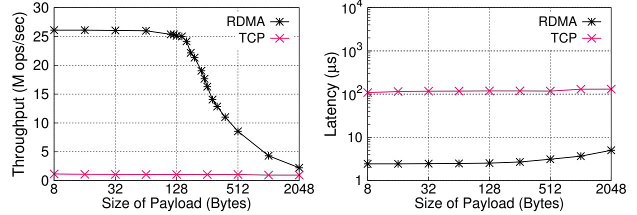

Figure 4: (a) The throughput and (b) the latency of random reads using one-sided RDMA READ and TCP/IP with different sizes of payload.

2.3 RDMA and Its Characteristics

Remote Direct Memory Access (RDMA) is a cross-node memory access technique with low-latency and low CPU overhead, due to complete bypassing of target OS kernel and/or CPU. RDMA provides both two-sided message passing interfaces like SEND/RECV Verbs as well as one-sided operations such as read, write and two atomic operations (fetch-and-add and compare-and-swap).

As noted in prior work [9, 32], one-sided operations are usually more efficient than its two-sided counterpart due to no CPU involvement in the target CPU. Further, as shown in Figure 4, the cost of one-sided RDMA operations are usually relatively insensitive to data sizes. For example, the latency only increases slightly even if the payload size increases to 2048 bytes and the throughput are almost the same till 128 Bytes for our RDMA NICs.

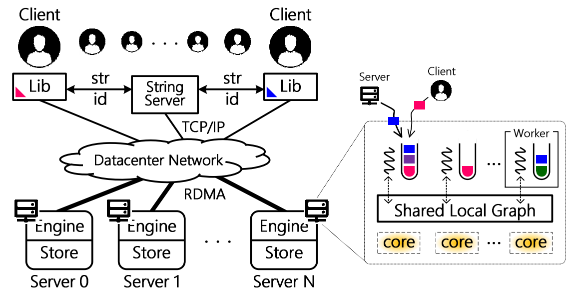

Figure 5: The architecture overview of Wukong.

3 Overview

Setting: Wukong assumes a cluster that is connected with high-speed, low-latency network with RDMA features. Wukong targets SPARQL queries over a large volume of RDF data; it scales by partitioning an RDF graph into a large number of shards across multiple machines. Wukong may create replicas for vertices to make sure each machine contains a complete subgraph of the input RDF graph, for better locality. Wukong also creates indexes vertices to assist queries. In each machine, Wukong employs a worker-thread model by running n worker threads atop n cores; each worker thread executes a query at a time.

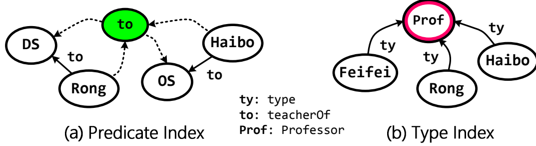

Figure 6: Two types of index vertex of Wukong.

Architecture: An overview of Wukong's architecture is shown in figure 5. Wukong follows a decentralized model in the server side, where each machine can directly serve clients' requests. Each client[5] contains a client library that parses SPARQL queries into a set of stored procedures, which are sent to the server side to handle the request. Alternatively, Wukong can also use a set of dedicated proxies to run the client-side library and balance client requests. To avoid sending long strings to the server and thus save network bandwidth, each string is first converted into a unique ID by the string server.

Each server consists of two separate layers: query engine and graph store. The query engine layer binds a worker thread on each core with a logical task queue to continuously handle requests. Graph store layer adopts an RDMA-friendly key/value store over distributed partitions of the RDF graph, which is shared by all of worker hashtable to support a partitioned global address space. Each machine threads on the same machine.

Query processing: Wukong is designed to provide low-latency to multiple concurrent queries from clients. The client or the proxy decides which server a request will be first sent to according to the request types. For a query starting with a constant vertex, Wukong sends the request to the server holding the master replica of the vertex. For a query starting with a set of vertices with a specific type or predicate, Wukong then send the request to all replicas of the corresponding index vertex.

Each query may be represented as a chain of sub-queries. Each machine handles a sub-query and then dispatches the remaining sub-queries to other machines when necessary. A sub-query may be executed immediately, or pushed into the task queue to be scheduled.

4 GRAPH-BASED RDF DATA MODELING

In this section, we provide a detailed description of the graph indexing, partitioning and storing strategies employed by Wukong, which is the base to sequentially and concurrently process SPARQL queries on RDF data.

4.1 Graph Model and Indexes

Wukong uses a directed graph to model and store RDF data, where each vertex corresponds to an entity in an RDF triple (*subject* or *object*) and each edge is labeled as a *predicate* and points from subjects to objects. As SPARQL queries may rely on retrieving a set of subjects/object vertices connected by edges with certain predicates, we provide two types of *index* vertices to assist such graph queries, as shown in Figure 6. To avoid confusion, we use the *normal* vertex to refer subjects and objects.

[5]The client may be not the end user but the front-end of Web service.

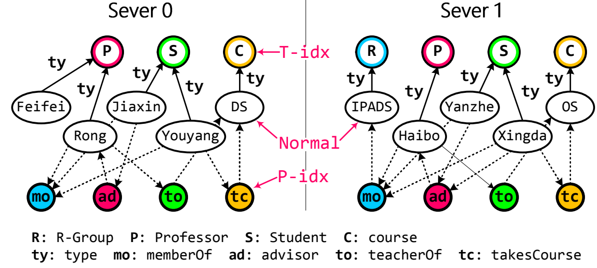

Figure 7: A hybrid graph partitioning on two servers.

For the query pattern with a certain predicate, like {?Y teacherOf ?Z} (see Q_2 in Figure 3), we propose the *predicate* index (P-idx) to maintain all of subjects and objects labeled with the particular predicate using its in and out edges respectively. For example, in Figure 6, a predicate index teacherOf (to) links to all normal vertices whose in-edges (DS and OS) or out-edges (Rong and Haibo) contain the label to. This corresponds to the PSO and POS indexes in triple store approaches.

Further, the special predicate type (ty) is used to group a set of subjects that belong to a certain type, like {?X type Prof} (see Q_1 in Figure 2). Therefore we treat the objects of such predicate as the *type* index (T-idx). For example, a type index Prof in Figure 6(b) maintains all normal vertices which are of the type of professors.

Unlike prior graph-based approaches that manage indexes using separate data structures, Wukong treats indexes as essential parts (vertices and edges) of RDF graph and also takes into consideration the graph partitioning and storing of them. This has two benefits. First, this eases query processing using graph exploration such that the graph exploration can directly start from an index vertex. Second, this makes it easy and efficient to distribute the indexes among multiple servers, as shown in the following sections.

4.2 Differentiated Graph Partitioning

One key step of supporting distributed query is partitioning a graph among multiple machines, while still preserving good access locality and enabling parallelism. We observe that complex queries usually involve a large number of vertices through a certain predicate or type, which should be executed on multiple machines to exploit parallelism.

Inspired by PowerLyra [8], Wukong adopts differentiated partitioning algorithms to normal and index vertex. As shown in Figure 7 Each normal vertex (e.g., DS) will be randomly assigned to one and only machine with all of edges by hashing the vertex ID. Note that the edges linked to predicate index (i.e. dotted arrows) will not be included in the edge list of normal vertices, since there is

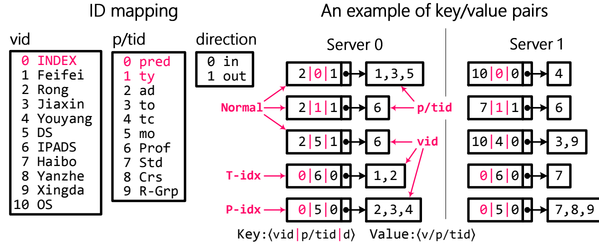

Figure 8: The design of predicate-based key/value store.

no need to find a predicate index vertex via normal vertices and this can save plenty of memory space. Different from normal vertex, each index vertex (e.g., takesCourse and Course) will be split and replicated to multiple machines with edges linked to normal vertices on the same machine. This naturally distributes the indexes and their workload among each machine.

4.3 Predicate-based RDMA-friendly Store

Similar to Trinity.RDF [39], Wukong uses a distributed key/value store to physically store the graph. However, unlike prior work that simply uses vertex ID (vid) as the key, and the in and out edge list (each element is a $\langle predicate, vid\rangle$ pair) as the value, Wukong uses a combination of the vertex ID (vid), predicate/type ID (p/tid) and in/out direction (d) as the key (in the form of $\langle vid, p/tid, d\rangle$), and the list of neighboring vertex IDs or predicate/type IDs as the value. The main observation is that a SPARQL query is usually concerned with querying upon partial neighboring vertices satisfying a particular predicate (e.g., X predicate ?Y). Therefore, missing the predicate and direction in key would lead to plenty of unnecessary computation cost and networking traffic. The finer-grained partitioning of vertices using predicates also makes it possible to build local predicate indexing, which corresponds to the PSO and POS indexes in triple store approaches.

To uniformly store normal and index vertices and adapt differentiated partitioning strategies, Wukong separates the ID mapping for vertex ID (vid) and predicate/type ID (p/tid). The ID 0 of vid (INDEX) is reserved for the index vertex, while the ID 0 and 1 of p/tid are reserved for the type and predicate indexes respectively. Figure 8 illustrates detailed cases on the sample graph. The key of normal vertex starts from a nonzero vid and relies on p/tid to distinguish different meanings of the value. The p/tid ID 0 and 1 represent the value as a list of predicate IDs and a type ID for the vertex respectively, otherwise the value is a list of normal vertices linked to the normal vertex with a certain predicate (p/tid). For example, the predicates labeled on out-edges of vertex Rong is represented as the key $\langle 2|0|1\rangle$, and the value $\langle 1,3,5\rangle$ means type, teacherOf and memberOf. While the type of vertex Rong is represented as the key $\langle 2|1|1\rangle$, and the value $\langle 6\rangle$ means Professor. The key of index vertex always starts from a zero vid, and linked to a list of local

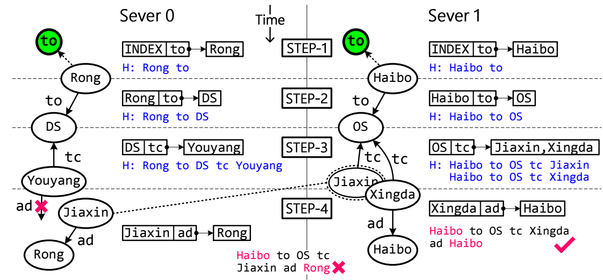

Figure 9: A sample of execution flow on Wukong

normal vertices. For example, all subjects of the predicate memberOf on Sever 0 (Rong, Jiaxin and Youyang) and Sever 1 (Haibo, Yanzhe and Xingda) are stored with the same key $\langle 0|5|0\rangle$ but different servers.

Finally, due to the goal of leveraging the advanced networking features such as RDMA, Wukong is built upon an RDMA-friendly distributed hashtable derived from DrTM-KV [32] and thus enjoys its nice features like RDMA-friendly cluster hashing and location-based cache. However, as the key/value store in Wukong is designed for query processing instead of transaction processing, we notably simplify the design by removing unnecessary metadata for checking consistency and supporting transactions.

5 QUERY PROCESSING

5.1 Basic Query Processing

An RDF query can be represented as a subgraph with free variables (i.e., not bound to specific subjects/objects yet). The goal of the query is to find bindings of specific subjects/objects to the free variables while respecting the subgraph pattern. However, it is well-known that using subgraph matching would be very costly due to the frequent yet costly joins [39]. Hence, like prior work [39], Wukong leverages graph exploration by walking the graph in specific orders according to each edge of the subgraph.

There are several cases for each edge in a graph query, depending on whether the subject, the predicate or the object are free variables. For the common cases where predicate is known but subject/object are free variables, Wukong can leverage the predicate index to start the graph exploration. Take the Q_2 in Figure 3 as an example, which aims at querying advisors, courses and students such that the advisor advises the student who also takes a course taught by the advisor. The query forms a cyclic subgraph containing three free variables. Wukong chooses an order of exploration according to some heuristics[6]. As shown in Figure 9, Wukong starts exploration from the teacherOf predicate (to). Since Wukong extends the graph with predicate indexes, it can

[6]like the selectivity of triples, a detailed cost-based optimization is out of the scope of this paper.

start exploration from the index vertex for teacherOf in each machine in parallel, whose neighbors contain Rong and Haibo in each server accordingly. In Step2, Wukong combines Rong and teacherOf to form a key to get the corresponding courses, which are {Rong to DS} and {Haibo to OS} accordingly. In Step3, Wukong continues to explore the graph from the course vertex for each tuple in parallel and tries to get all students that take the course. Thanks to the differentiated graph partitioning, there is no communication through Step1-3. In Step4, Wukong leverages the constraint information to filter out non-matching results to get the final result.

For (rare) cases where the predicate is unknown, Wukong starts graph exploration from a constant vertex (in cases where either subject or object is known) with a p/tid 1 (Pred). The value is the list of predicates associated with the vertex, and then Wukong iterates them one by one. The remaining process is similar to those described above.

5.2 Full-history Pruning

Note that there could be tuples that should be filtered out during the graph exploration. For example, since there is no expected advisor predicate (ad) for Youyang, the related tuples should be filtered out to minimize redundant computation and communication. Further, in Step 4, as Jiaxin's advisor is Rong instead of Haibo, the graph exploration path also should be pruned as well.

Prior graph-exploration strategies [39] usually use a one-step pruning approach by leveraging the constraint in the immediately prior step to filter some unnecessary information out. In the final step, it leverages a single machine to aggregate and conduct a final join over the results to filter out non-matching results. However, recent study [12, 22] found that, the final join can easily become the bottleneck of a query since all results need to be aggregated into a single machine for joining.

Instead, Wukong adopts a *full-history pruning* approach such that Wukong passes the full exploration history to the next step across machines. The main observation is that, the cost of RDMA operations are insensitive to the size of the payload when it is smaller than a certain size (e.g., 256 Bytes) [10, 32]. Besides, the steps in an RDF query are usually not many (i.e., less than 10) and thus there won't be too much information carried even for the final few steps. Hence, the cost remains the same for passing more history information across machines since each history item only contains subject/object/predicate IDs and thus won't be very large even for a long path of queries. Further, passing full history locally is roughly like adding more query variables in each step and thus the cost is negligible. As shown in Figure 9, Wukong passes {Rong to}, {Rong to DS} and {Rong to DS tc Youyang} in locally Sever 0 in each step; Youyang

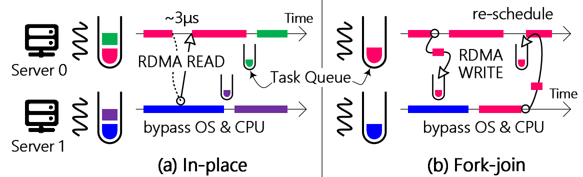

Figure 10: A sample of (a) in-place and (b) fork-join execution.

can be simply pruned without using history information due to no expected predicate (ad). Sever 0 can leverage the full history ({Haibo to OS tc Jiaxin}) from Server 1 to prune Jiaxin as Jiaxin's advisor is not Haibo.

As Wukong has the full history during graph exploration, there is no need of a final join to filter out non-matching results. Thought it appears that Wukong may bring additional network traffic when fetching cross-machine history, the fact that Wukong can prune non-matching results early may save network traffic as well.

5.3 Migrating Execution or Data

During the graph exploration process, there will be different tradeoffs on whether migrating execution or data. Wukong supports *in-place* and *fork-join* executions accordingly. For a query step, if only a few vertices need to fetch from remote machines, Wukong uses in-place execution mode that synchronously leverages one-sided RDMA READ to directly fetch vertices from remote machines, as shown in Figure 10(a). Using one-sided RDMA READ can enjoy the benefit of bypassing remote CPU and OS. For example, in Figure 9, Server 1 can directly read the advisor of Jiaxin with RDMA READ, and locally generate ({Haibo to OS tc Jiaxin ad RONG}).

For a query step, if many vertices may be fetched, Wukong leverages a *fork-join* execution mode that asynchronously splits the following query computation into multiple sub-queries running on remote machines. Wukong leverages one-sided RDMA WRITE to directly push a sub-query with full history to the task queue of a remote machine, as shown in Figure 10(b). This can also be done without bothering remote CPU and OS. For example, in Figure 9, Server 1 can send a sub-query with the full history ({Haibo to OS tc Jiaxin}) to Server 0. Server 0 will locally execute the sub-query to generate ({Haibo to OS tc Jiaxin ad Rong}). Note that, depending on the sub-query, the target machine may further do a fork-join operation to remote machines, forming a query tree. Each fork point then joins its forked sub-queries and returns the results to the parent fork point.

Since the cost of RDMA operations are insensitive to the size of the payload, for each query step, Wukong makes a decision on execution mode in runtime according to the number of RDMA operations ($|N|$). Each server will decide individually. For fork-join, $|N|$ is twice the number of servers. For in-place, $|N|$ is equal to the number of required vertices. Wukong simply uses a

```
1 int next = 1

 OBLIGER()
2  s = state[(tid+next)%N]
3  q = NULL
4  s.lock()
5  if (s.cur == tid
6   || s.end < now)
7    s.cur = tid;
8    s.end = now + T
9    next++
10   q = s.dequeue()
11 s.unlock()
12 return q
```

```
 SELF()
13 s = state[tid]
14 s.lock()
15 s.cur = tid
16 s.end = now + T
17 next = 1
18 q = s.dequeue()
19 s.unlock()
20 return q

 NEXT_QUERY()
21 if (q = OBLIGER())
22   return q
23 return SELF()
```

Figure 11: The pseudo-code of worker-obliger algorithm.

heuristic fixed threshold according to the setting of cluster. Further, some vertices have a significant large number of edges with the same predicate, resulting in slower RDMA READ due to oversize payload. Wukong can label such vertices associated with the predicate to enforce using fork-join mode when partitioning the RDF graph.

5.4 Handling Concurrent Queries

Depending on the complexity and selectivity, the latency (i.e., execution time) of a query could vary significantly. For example, the latency differences among seven queries in LUBM [2] can reach around 5,000X (0.2ms and 1040ms for L5 and L7 queries accordingly). Hence, dedicating an entire cluster for a single query, as done in prior approaches [39, 12], is not cost-effective.

Wukong is designed to handle a massive number of queries concurrently while trying to parallelize a single query to reduce the query latency. The difficulty is that, given the significantly varied query latencies, how to minimize intra-query interference while providing good utilization of resources, e.g., a lengthy query should not significantly extend the latency of a quick query.

The online sub-query decomposition and the dynamic execution mode switching serve as a keystone to support massive queries in parallel. Specifically, Wukong uses a private FIFO queue to schedule queries for each worker thread, which works well for small queries. However, if there is a lengthy query, it will monopolize the worker thread and impose queuing delays on the execution of small waiting queries. This will incur much higher latency than necessary. Worse even, a lengthy query with multi-threading enabled (section 6) may monopolize the entire cluster.

To this end, Wukong uses a *worker-obliger* work stealing algorithm for multiple workers in each machine, as shown in Figure 11. Each worker is designated to oblige next few neighboring workers in case they are busy with processing a lengthy (sub-)query. After finishing a (sub-)query, a worker first checks a neighboring worker in turn if its (sub-)query has finished in time (e.g., `s.end < now`). If not, that worker might be handling a lengthy query and thus its following up queries may be delayed.

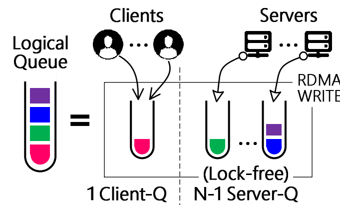

Figure 12: The logical task queue in Wukong.

In this case, this obliging worker steals one query from that worker's queue to handle. After obliging its neighboring workers (if needed), the worker will not forget to handle its own queries by dequeuing from its own queue.

Note that, when all workers can handle their queries within a threshold (i.e., `T`), each worker only needs to handle queries in its own queue. The checking code is also very lightweight and the state lock (i.e., `s.lock()`) won't be contended as there will only at most two workers (i.e., SELF and OBLIGER) may try to acquire the lock. It could be possible that an obliger may get sucked in handling a lengthy query for others; in this case, another worker may oblige him similarly.

6 IMPLEMENTATION

The Wukong prototype comprises around 6,000 lines of C++ code. It currently runs atop an RDMA-capable cluster. This section describes some implementation issues.

Task queues: Wukong binds a worker thread on each core with a logical private task queue, which is used by both clients and worker threads on other servers to submit (sub-)queries. Wukong leverages RDMA operations (especially one-sided RDMA) to accelerate the communication among worker threads; however, the clients may still connect servers using general interconnects.

The logical queue per thread in Wukong consists of one client queue (Client-Q) and multiple server queues (Server-Q). For the client queue, Wukong follows traditional concurrent queue to serve the queries from many clients. But due to the lack of expressiveness of one-sided RDMA operations, implementing RDMA-based concurrent queue may incur large overhead. On the contrary, using separate task queues for each worker threads of each remote machine may exponentially increase the number of queues. Fortunately, we observe that there is no need to allow all worker threads on a remote machine sending queries to all local worker threads. To remedy this, Wukong only provides a one-to-one mapping between the work threads on different machines, as shown in Figure 12. This can avoid not only the burst of task queues but also complicated concurrent mechanisms.

Launching query: To launch a query, the start point of a query can be a normal vertex (e.g., {?X memberOf IPADS}) or a predicate or type index (e.g., {?X teacherOf ?Y}). Since the index vertex is replicated to multiple servers, Wukong allows the client library to send the

Table 1: A collection of real-life and synthetic datasets.

Dataset	#Triples	#Subjects	#Objects	#Predicates
LUBM-10240	1,410M	222M	165M	17
WSDTS	109M	5.2M	9.8M	86
DBPSB	15M	0.3M	5.2M	14,128
YAGO2	190M	10.5M	54.0M	99

same query to all servers such that the query can be distributed from the beginning. However, distributed execution may not be worthwhile for a low-degree index vertex. Therefore, Wukong will decide whether replicas of an index vertex need to process the query or not when partitioning the RDF graph. For low-degree index vertices, the master will process the query alone by aggregating data from replicas through one-sided RDMA READ, and the replicas will simply discard queries. For high-degree index vertices, both the master and replicas will individually process the query on local graph.

Multi-threading: By default, Wukong processes a (sub-)query using only a single thread on each server. To reduce latency of a query, Wukong also allows running a time-consuming query with multiple threads on each server, at the requests of clients. A worker thread received the multi-threaded (MT) query will invite other worker threads on the same server to process the query in parallel. Wukong adopts a data-parallel approach to automatically parallelize the query after the first graph exploration. Each worker thread will individually process the query on a part of subgraph. Note that the maximum number of participants for a query is claimed by the client, but finally restricted by an MT threshold of server.

7 EVALUATION

7.1 Experimental Setup

Hardware configuration: All evaluations were conducted on a rack-scale cluster with 6 machines. Each machine has two 10-core Intel Xeon E5-2650 v3 processors and 128GB of DRAM. We disabled hyperthreading on all machines. Each machine is equipped with a ConnectX-3 MCX353A 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps InfiniBand Switch. All machines run Ubuntu 14.04 with Mellanox OFED v3.0-2.0.1 stack.

In all experiments, We reserve two cores on each processor to generate requests for all machines to avoid the impact of networking between clients and servers as done in prior OLTP work [32, 10, 30, 29]. For a fair comparison, we measure the query execution time by excluding the cost of literal/ID mapping. All experimental results are the average of five runs.

Benchmarks: We use two synthetic and two real-life datasets, as shown in Table 1. The synthetic datasets are the Lehigh University Benchmark (LUBM) [2] and the Waterloo SPARQL Diversity Test Suite (WSDTS) [3]. For LUBM, we generate 5 datasets with different sizes

Table 2: The query performance (msec) on a single server.

LUBM 2560	Wukong	RDF-3X (warm)	RDF-3X (cold)	BitMat (warm)	BitMat (cold)
L1	752	2.3E5	2.5E5	abort	abort
L2	146	4,494	1.1E5	36,256	38,730
L3	316	3,675	4,817	752	1,439
L4	0.19	2.2	276	55,451	57,242
L5	0.11	1.0	180	52	101
L6	0.57	37.5	465	487	696
L7	1,325	9,927	1.3E5	19,323	22,295
Geo. Mean	18	441	6,319	–	–

Table 3: The query performance (msec) on a 6-node cluster.

LUBM 10240	Wukong	TriAD	TriAD-SG (200K)	Trinity .RDF	SHARD
L1	516	2,110	1,422	12,648	19.7E6
L2	88	512	695	6,081	4.4E6
L3	260	1,252	1,225	8,735	12.9E6
L4	0.48	3.4	3.9	5	10.6E6
L5	0.18	3.1	4.5	4	4.2E6
L6	0.88	63	4.6	9	8.7E6
L7	1,040	10,055	11,572	31,214	12.0E6
Geo. Mean	19	190	141	450	9.1E6

using the generator v1.7 in NT format. For queries, we use the benchmark queries published in Atre et al. [5], which were widely used by may distributed RDF systems [12, 39, 16]. WSDTS publishes a total of 20 queries in four categories. The real-life datasets are the DBpedia's SPARQL Benchmark (DBPSB) [1] and YAGO2 [13]. For DBPSB, we choose 5 queries provided by its official website. YAGO2 is a semantic knowledge base, derived from Wikipedia, WordNet and GeoNames. We follow the queries defined in H_2RDF+ [22].

Comparing targets: We compare the query performance of Wukong against several state-of-the-art systems. 1) centralized systems: RDF-3X [19] and BitMat [5]; 2) distributed systems: TriAD [12], Trinity.RDF [39] and SHARD [25]. Since Trinity.RDF is not publicly available and TriAD reported superior performance over it, we only directly compare the results published in their paper [39] with the same workload.

7.2 Single Query Performance

We first study the performance of Wukong for a single query using the LUBM dataset.

For a fair comparison to centralized systems, we also run Wukong on a single machine. Since both RDF-3X and BitMat are disk-based, we report both warmcache and cold-cache time. As shown in Table 2, Wukong outperforms the on-disk performances of RDF-3X and BitMat by more than two orders of magnitude, except for L3. L3 has an empty final result even with huge intermediate results and thus there is no significant performance difference. For in-memory performance, Wukong still outperforms RDF-3X and BitMat by one order of magnitude, due to fast graph exploration for simple queries and efficient multi-threading for complex queries.

We further compare Wukong with distributed systems with multi-threading enabled in Table 3. For selective queries (L4, L5 and L6), Wukong outperforms TriAD by

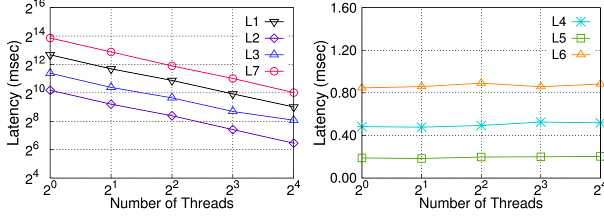

Figure 13: The latency of queries in group (I) and (II) on LUBM-10240 with the increase of threads.

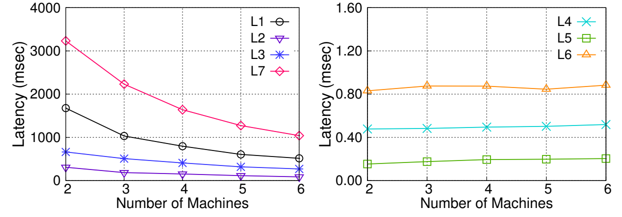

Figure 14: The latency of queries in group (I) and (II) on LUBM-10240 with the increase of machines.

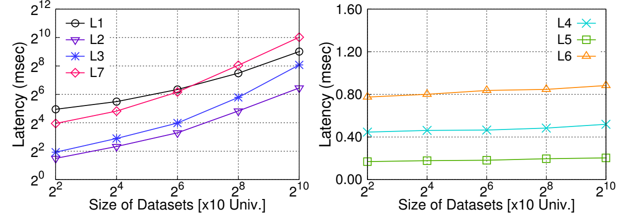

Figure 15: The latency of queries in group (I) and (II) with the increase of LUBM datasets (160-10240).

up to 71.2X (from 7.2X) due to the in-place execution with one-sided RDMA READ. For non-selective queries (L1, L2, L3 and L7), Wukong still outperforms TriAD by up to 9.7X (from 4.1X), thanks to the fast graph exploration with finer-grained partitioning and full-history pruning. The join-ahead pruning with summary graph (SG) improves the performance of TriAD, especially for L1 and L6, while Wukong still outperforms the average (geometric mean) latency of TriAD-SG by 7.4X (ranging from 2.8X to 25.4X). Compared to Trinity.RDF, which also uses graph-exploration strategy, the improvement of Wukong is at least one order of magnitude (from 10.2X to 69.4X), thanks to the full-history pruning that avoids redundant computation and communication as well as the time-consuming final join. Note that the result of Trinity.RDF is evaluated on a cluster with similar interconnects and twice the number of servers. SHARD is several orders of magnitude slower than other systems since it randomly partitions the RDF data and employs Hadoop as a communication layer for handling queries.

7.3 Scalability

We evaluate the scalability of Wukong in three aspects by scaling the number of threads, the number of servers, and the size of dataset accordingly. We categorize seven queries on LUBM dataset into two groups according to the sizes of their intermediate and final results as done in prior work [39]. Group (I): L1, L2, L3, and L7; the results of such queries increase with the growing of dataset. Group (II): L4, L5, L6; such queries are quite selective and produce fixed-size results regardless of the data size.

Scale-up: We first study the performance impact of multi-threading on LUBM-10240 using fixed 6 servers. Figure 13 shows the latency of queries on a logarithmic scale with the logarithmic increase of threads. For group (I), the speedup of Wukong ranges from 9.9X to 14.3X with the increase of threads from 1 to 16. For group (II), since the queries just involve a small subgraph and are not CPU-intensive, Wukong always adopts a single thread for the query and provides a stable performance.

Scale-out: We also evaluate the scalability of Wukong with respect to the number of servers. Note that we omit the evaluation on a single server as LUBM-10240 (amounting to 230GB in raw NT format) cannot fit into memory. Figure 15(a) shows a linear speedup of Wukong for group (I) ranging from 2.46X to 3.54X, with the increase of servers from 2 to 6. It implies Wukong can efficiently utilize the parallelism of a distributed system by leveraging fork-join execution mode. For group (II), since the intermediate and final results are relatively small and fixed-size, using more machines does not improve the performance as expected, but the performance is still stable by using in-place execution to restrict the network overhead.

Data size: We further evaluate Wukong with the increase of dataset size from LUBM-40 to LUBM-10240 while keeping the number of threads and servers fixed. As shown in Figure 15, For group (I), Wukong scales quite well with the growing of dataset, due to efficiently passing full history and the elimination of the final join. For group (II), Wukong can achieve stable performance regardless of the increasing dataset size, due to the in-place execution with one-sided RDMA READ.

7.4 Throughput of Mixed Workloads

Unlike prior graph-based RDF stores that are only designed to handle one query at a time, Wukong is also designed to provide high throughput such that it can handle hundreds of thousands of concurrent queries per second. Therefore, we build emulated clients and various mixture workloads to study the behavior of RDF stores serving concurrent queries.

For Wukong, each server runs up to 4 emulated clients on dedicated cores. All clients will send as many queries as possible periodically until the throughput saturated. For TriAD, a single client will send queries one by one since it only can handle one query at a time.

We first use a mixture workload consisting of 6 classes of queries[7], all of which disable multi-threading. The

[7] The templates of 6 classes of queries are based on group (II) queries (L4, L5 and L6) and three additional queries from official website (A1, A2 and A3).

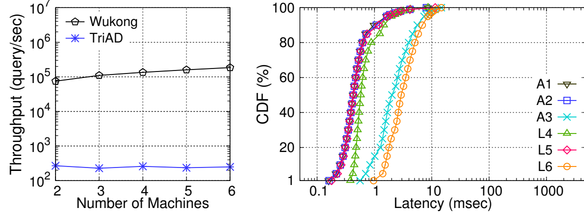

Figure 16: (a) The throughput of a mixture of queries with the increase of machines, and (b) the CDF of latency for 6 classes of queries on 6 machines.

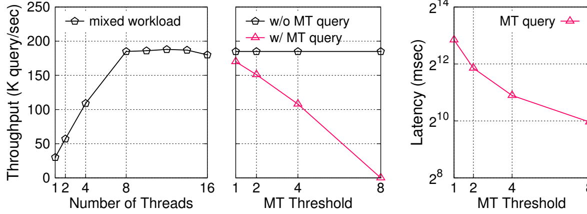

Figure 17: (a) The throughput of a mixture of queries with the increase of threads, (b) the throughput w/ and w/o multi-threaded (MT) queries using fixed 8 threads, and (c) the average latency of multi-threaded (MT) queries.

query in each class has a similar behavior except that the start point is randomly selected from the same type of vertices (e.g., Univ0, Univ1, etc.). The distribution of query classes follows the reciprocal of their average latency. As shown in Figure 16, Wukong achieves a peak throughput of 185K queries/second on 6 machines (75K queries/second on 2 machines), which is at least two orders of magnitude higher than TriAD (from 278X to 740X). Under the peak throughout, the geometric mean of 50th (median) and 99th percentile latency is just 0.80 and 5.90 milliseconds respectively.

Multi-threading query: To further study the impact of enabling multi-threading (MT) for time-consuming queries. We dedicate a client to continually send MT queries (i.e. L1) and configure Wukong with different MT thresholds. Since the throughout does not scale beyond 8 threads due to the bottleneck of networking (see Figure 17(a)), we use 8 worker threads in experiment. As shown in Figure 17(b) and (c), with the increase of the MT threshold, both the throughput of Wukong and the time of interference (the latency of MT query) will degrade. For example, under threshold 4, Wukong can still perform 108K query/sec and the average latency of MT query is about 1,901 msec.

Worker-obliger mechanism: The MT query will also influence the latency of the other small waiting queries. Figure 18(a) show the CDF of latency for 6 classes of non-MT queries. The 80th percentile latency increases at least two orders of magnitude and the 99th percentile latency reaches several thousands of msec. Relying on worker-obliger work stealing design, as shown in Figure 18(b), Wukong can recover the latency and meanwhile preserving the throughput.

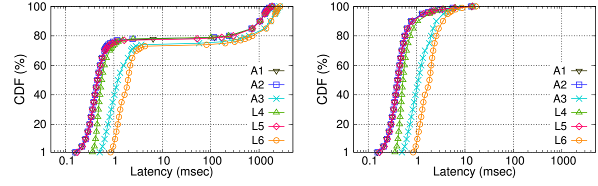

Figure 18: The CDF of latency for 6 classes of queries on 6 machines (a) w/o and (b) w/ worker-obliger mechanism. Each server uses fixed 8 threads (threshold=4).

Table 4: A performance comparison (msec) of w/ or w/o predicate-based key/value store (PBS) on LUBM-10240

LUBM	L1	L2	L3	L4	L5	L6	L7
w/o PBS	1,265	95	270	0.53	0.21	1.37	1,072
w/ PBS	516	88	260	0.48	0.18	0.88	1,040

7.5 Predicate-based Graph Store

Predicate-based graph store (PBS) adopts the finer-grained partitioning of vertices by predicates. Table 4 compares the latency of queries on LUBM-10240 with and without PBS. For query L1, PBS can achieve 2.45X improvement, since the required entities (i.e., Universities) have a large number of data with different predicates. For other queries, the improvement of PBS ranges from 1.03X to 1.56X.

Table 5: A performance comparison (msec) of various execution mode on LUBM-10240

LUBM	L1	L2	L3	L4	L5	L6	L7
In-place	26,065	88	262	0.51	0.21	2.39	16,492
Fork-join	1,183	90	269	0.79	0.55	1.21	1,080
Dynamic	516	88	260	0.48	0.18	0.88	1,040

7.6 In-place vs. Fork-join Execution

To study the benefit of dynamic choice between in-place and fork-join execution modes, we configure Wukong with a fixed mechanism (i.e., in-place or fork-join). Table 5 shows the latency of queries with various execution modes. In-place execution is better for queries L4 and L5, while fork-join execution is better for query L7. In addition, L2 and L3 are not sensitive to the choice of execution modes. L1 and L6 are relatively special, in which different steps require different execution modes for achieving optimal performance. Wukong can always choose the best execution mode in runtime and outperform in-place and fork-join by up to 50.5X and 2.8X.

7.7 Other Datasets

We further study the performance of Wukong and TriAD over more other synthetic and real-life datasets. Note that we do not provide the performance of TriAD-SG because the hand-tuned parameter of summary graph is not known and it only improves performance in few cases.

WSDTS: We first compare the performance of TriAD and Wukong over WSDTS dataset using 20 diverse queries, which are classified into linear (L), star (S), snowflake (F) and complex (C). Table 6 shows the geometric mean of latency for various query classes.

Table 6: The latency (msec) of queries on WSDTS

WSDTS	L1-L5 (Geo. M)	S1-S7 (Geo. M)	F1-F5 (Geo. M)	C1-C3 (Geo. M)
TriAD	4.5	5.3	17.5	36.6
Wukong	1.0	1.1	4.1	10.3

Wukong always outperforms TriAD by up to 20.0X (from 1.6X). For L1, L3, S1, S7 and F5, Wukong is at least one order of magnitude faster than TriAD since the queries are quite selective and appropriate for graph exploration. For only two queries, F1 and C3, the improvement of Wukong is less than 2.0X.

Table 7: The latency (msec) of queries on DBPSB

DBPSB	D1	D2	D3	D4	D5	Geo. Mean
TriAD	4.93	4.10	5.56	7.68	3.51	4.97
Wukong	1.75	0.48	0.41	3.70	1.14	1.16

DBPSB: Table 7 shows the performance of five representative queries on DBPSB, which is a relative small real-life dataset, but has quite more predicates. Wukong outperforms TriAD by at least 2X (up to 13.6X), and the improvement of geometric mean reaches 4.3X. For D2 and D3, the speedup reaches 8.6X and 13.6X respectively since the queries are relatively selective.

Table 8: The latency (msec) of queries on YAGO2

YAGO2	Y1	Y2	Y3	Y4	Geo. Mean
TriAD	1.13	2.14	68,841	6,193	179
Wukong	0.12	0.17	38,571	3,501	41

YAGO2: Table 8 compares the performance of TriAD and Wukong on a large real-life dataset YAGO2. For the simple queries, Y1 and Y2, Wukong is one order of magnitude faster than TriAD due to fast in-place execution. For the complex queries, Y3 and Y4, Wukong can still notably outperforms TriAD by about 1.8X due to full-history pruning and RDMA-friendly task queues.

8 Related Work

RDF query over triple and relational store: There have been a large number of triple-based RDF stores that use relational approaches to storing and indexing RDF data [19, 20, 4, 33, 26, 7]. Since join is expensive and a key step for query processing in such triple stores, they perform various query optimizations including heuristic optimizations [19], join-ordering exploration [19], join-ahead pruning [20], graph summarization [40] and query caching [24]. Specially, TriAD [12] is a most recent distributed in-memory RDF engine that leverages join-ahead pruning and graph summarization with asynchronous message passing for parallelization. SHAPE [16] is a distributed engine upon RDF-3X by statically replicating and prefetching data. As shown in prior work [39], graph exploration can avoid many redundant immediate results generated during expensive join operations and thus typically deliver better performance. There is also a recent study, SQLGraph [28], that explores the idea of using a relational store to store the RDF data, but process RDF queries as a graph store. Their focus, however, is on query rewriting and schema refinement; furthermore, they want to support heavyweight, ACID-style transactions (for updates). As a result, SQLGraph is a centralized approach and has different objectives from our study.

RDF query over graph store: There is an increasing interest in using native graph model to store and query RDF data [5, 36, 38, 39]. BitMat [5], gStore [40] and TripleBit [38] are centralized graph stores with sophisticated indexes to improve query performance. Sedge [37] is a distributed SPARQL query engine based on a simple Pregel implementation, which tries to minimize the inter-machine communication by group-based communication. The most related work is Trinity.RDF [39], a distributed in-memory RDF store that leverages graph exploration to process queries. Wukong's design centers around the usage of fast interconnect with RDMA features to allow fast graph exploration. Wukong also introduces novel graph-based indexes as well as differentiated graph partitioning and query processing to improve the overall system performance.

RDF query over MapReduce: Several distributed RDF systems are built atop existing frameworks like MapReduce [23, 22, 25, 27], e.g., H_2RDF [23, 22] and SHARD [25]. PigSPARQL [27] maps SPARQL operations into PigLatin [21] queries, which in turn is translated into MapReduce programs. However, due to the lack of efficient iterative computation support, MapReduce-based computation is usually sub-optimal for SPARQL execution, as shown in prior work [12, 39].

RDMA-centric stores: The low latency and high throughput of RDMA-based networking stimulates much work on RDMA-centric key/value stores [17, 15], OLTP platforms [32, 10] and general graph analytics engines [34, 14]. Specially, GraM [34] is an efficient and scalable graph analytics engine that leverages multicore and RDMA to provide fast batch-oriented graph analytics. However, handling SPARQL queries is significantly different from general graph analytics and thus Wukong can hardly benefit from the design of GraM. Further, Wukong is designed to handle highly concurrent queries while GraM is designed to handle one offline graph-analytics task at a time.

9 Conclusion and Future Work

This paper describes Wukong, a distributed in-memory RDF store that leverages RDMA-based graph exploration to support fast and concurrent RDF queries. Wukong significantly outperforms state-of-the-art systems and can process a mixture of small and large queries at 185,000 queries/second on a 6-note cluster. Currently, we use simple heuristics to generate query plans, our future work may include a cost-based optimal query planning algorithm for further performance optimization.

REFERENCES

[1] Dbpedias sparql benchmark. http://aksw.org/Projects/DBPSB.

[2] Swat projects - the lehigh university benchmark (lubm). http://swat.cse.lehigh.edu/projects/lubm/.

[3] Waterloo sparql diversity test suite (wsdts). https://cs.uwaterloo.ca/ galuc/wsdts/.

[4] ABADI, D. J., MARCUS, A., MADDEN, S. R., AND HOLLENBACH, K. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB JournalThe International Journal on Very Large Data Bases 18*, 2 (2009), 385–406.

[5] ATRE, M., CHAOJI, V., ZAKI, M. J., AND HENDLER, J. A. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 41–50.

[6] BIO2RDF CONSORTIUM. Bio2rdf: Linked data for the life science. http://bio2rdf.org/, 2014.

[7] BORNEA, M. A., DOLBY, J., KEMENTSIETSIDIS, A., SRINIVAS, K., DANTRESSANGLE, P., UDREA, O., AND BHATTACHARJEE, B. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 121–132.

[8] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 1:1–1:15.

[9] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14, USENIX Association, pp. 401–414.

[10] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP'15, ACM, pp. 54–70.

[11] GOOGLE INC. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[12] GURAJADA, S., SEUFERT, S., MILIARAKI, I., AND THEOBALD, M. Triad: A distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 289–300.

[13] HOFFART, J., SUCHANEK, F. M., BERBERICH, K., LEWIS-KELHAM, E., DE MELO, G., AND WEIKUM, G. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference Companion on World Wide Web* (New York, NY, USA, 2011), WWW '11, ACM, pp. 229–232.

[14] HONG, S., DEPNER, S., MANHARDT, T., VAN DER LUGT, J., VERSTRAATEN, M., AND CHAFI, H. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 58:1–58:12.

[15] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), SIGCOMM'14, ACM, pp. 295–306.

[16] LEE, K., AND LIU, L. Scaling queries over big rdf graphs with semantic hash partitioning. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1894–1905.

[17] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference* (2013), pp. 103–114.

[18] NATIONAL CENTER FOR BIOTECHNOLOGY INFORMATION. Pubchemrdf. https://pubchem.ncbi.nlm.nih.gov/rdf/, 2014.

[19] NEUMANN, T., AND WEIKUM, G. Rdf-3x: A risc-style engine for rdf. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 647–659.

[20] NEUMANN, T., AND WEIKUM, G. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 627–640.

[21] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1099–1110.

[22] PAPAILIOU, N., KONSTANTINOU, I., TSOUMAKOS, D., KARRAS, P., AND KOZIRIS, N. H2rdf+: High-performance distributed joins over large-scale rdf graphs. In *2013 IEEE International Conference on Big Data* (2013), IEEE BigData '13, IEEE, pp. 255–263.

[23] PAPAILIOU, N., KONSTANTINOU, I., TSOUMAKOS, D., AND KOZIRIS, N. H2rdf: Adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web* (New York, NY, USA, 2012), WWW '12 Companion, ACM, pp. 397–400.

[24] PAPAILIOU, N., TSOUMAKOS, D., KARRAS, P., AND KOZIRIS, N. Graph-aware, workload-adaptive sparql query caching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1777–1792.

[25] ROHLOFF, K., AND SCHANTZ, R. E. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications* (New York, NY, USA, 2010), PSI EtA '10, ACM, pp. 4:1–4:5.

[26] SAKR, S., AND AL-NAYMAT, G. Relational processing of rdf queries: A survey. *SIGMOD Rec. 38*, 4 (June 2010), 23–28.

[27] SCHÄTZLE, A., PRZYJACIEL-ZABLOCKI, M., AND LAUSEN, G. Pigsparql: Mapping sparql to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management* (2011), ACM, p. 4.

[28] SUN, W., FOKOUE, A., SRINIVAS, K., KEMENTSIETSIDIS, A., HU, G., AND XIE, G. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1887–1901.

[29] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD'12, ACM, pp. 1–12.

[30] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP'13, ACM, pp. 18–32.

[31] WEAVER, J., AND TARJAN, P. Facebook linked data via the graph api. *Semantic Web 4*, 3 (2013), 245–250.

[32] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 87–104.

[33] WEISS, C., KARRAS, P., AND BERNSTEIN, A. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 1008–1019.

[34] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[35] WU, W., LI, H., WANG, H., AND ZHU, K. Q. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 481–492.

[36] YANG, S., YAN, X., ZONG, B., AND KHAN, A. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 517–528.

[37] YANG, S., YAN, X., ZONG, B., AND KHAN, A. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 517–528.

[38] YUAN, P., LIU, P., WU, B., JIN, H., ZHANG, W., AND LIU, L. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow. 6*, 7 (May 2013), 517–528.

[39] ZENG, K., YANG, J., WANG, H., SHAO, B., AND WANG, Z. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, VLDB Endowment, pp. 265–276.

[40] ZOU, L., MO, J., CHEN, L., ÖZSU, M. T., AND ZHAO, D. gstore: Answering sparql queries via subgraph matching. *Proc. VLDB Endow. 4*, 8 (May 2011), 482–493.