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

Contacts: {rongchen, haibochen}@sjtu.edu.cn

Abstract

Many public knowledge bases 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 provide highly concurrent and low-latency queries over large data sets. Wukong is novel in three ways. First, Wukong provides an RDMA-friendly distributed key/-value store that provides differentiated encoding and fine-grained partitioning of graph data to reduce RDMA transfers. Second, Wukong leverages full-history pruning to avoid the cost of expensive final join operations, based on the observation that the cost of one-sided RDMA operations is largely oblivious to the payload size to a certain extent. Third, countering conventional wisdom of preferring migration of execution over data, Wukong seamlessly combines data migration for low latency and execution distribution for high throughput by leveraging the low latency and high throughput of one-sided RDMA operations, and proposes a worker-obliger model for efficient load balancing.

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]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 extremely fast speed (21,675 kilometers in one somersault) and the ability to fork himself to do massive multi-tasking, we term our system as Wukong. The source code and a brief instruction on how to install Wukong is available at http://ipads.se.sjtu.edu.cn/projects/wukong.

1 Introduction

Many large datasets are continuously 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 [20] and a number of public knowledge bases including DBpedia [1], Probase [51], PubChem-RDF [32] and Bio2RDF [7]. 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, it is highly demanding that an RDF store provides low latency and high throughput over highly concurrent queries. In response, much recent research has been devoted to developing scalable and high performance systems to index RDF data and process SPARQL queries. Early RDF stores like RDF-3X [33], SW-Store [8], HexaStore [49] usually use a centralized design, while later designs such as TriAD [21], Trinity.RDF [54], H_2RDF [38] and SHARD [40] use a distributed store in response to the growing data sizes.

An RDF dataset is essentially a labeled, directed multigraph. Hence, it may be either stored as a set of triples as elements in relational tables (i.e., a triple store) [33, 21, 38, 53], or managed as a native graph (i.e., a graph store) [9, 58, 52, 54]. Prior work [54] shows that while using a triple store may enjoy query optimizations designed for relational database queries, query processing intensively relies on join operations over potentially large tables, which usually generates huge redundant intermediate data. Besides, using relational tables to store triples may limit the generality such that existing systems can hardly support general graph queries over RDF data such as reachability analysis and community detec-

[2]A recursive acronym for SPARQL Protocol and RDF Query Language.

tion [44].

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 concurrent queries per second. The key techniques of Wukong are centered around using one-sided RDMA to provide fast and concurrent graph exploration.

RDMA-friendly Graph Model and Store (§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 parts of the graph. To partition and distribute data among multiple machines, Wukong applies a differentiated partition scheme [13] to embrace both locality (for normal vertices) and parallelism (for index vertices) during query processing. Based on the observation that RDF queries only touch a small subset of graph data (e.g., a subset of vertices and/or a subset of a vertex's data), Wukong further incorporates predicate-based finer-grained vertex decomposition and stores the decomposed graph data into a refined, RDMA-friendly distributed hashtable inherited from DrTM-KV [48] to reduce RDMA transfers.

RDMA-based Full-history Pruning (§5.2). Being aware of the cost-insensitivity of one-sided RDMA operations with respect to data size, Wukong leverages *full-history pruning* such that it can precisely prune unnecessary intermediate data. Consequently, Wukong can avoid the costly centralized final join on the results aggregated from multiple machines.

RDMA-based Query Distribution (§5.3). Depending on the selectivity and complexity of queries, Wukong decomposes a query into a sequence of sub-queries and handles multiple independent sub-queries simultaneously. For each sub-query, Wukong adopts an 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. To prevent large queries from blocking small queries when handling concurrent queries, Wukong provides a latency-centric work stealing scheme (namely *worker-obliger model*) 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 state-of-the-art centralized (e.g., RDF-3X and BitMat) and distributed (e.g., TriAD and Trinity.RDF) systems. An evaluation using a mixture of queries on LUBM [3] shows that Wukong can achieve up to 269K queries per second on 6 machines with 0.80 milliseconds median latency.

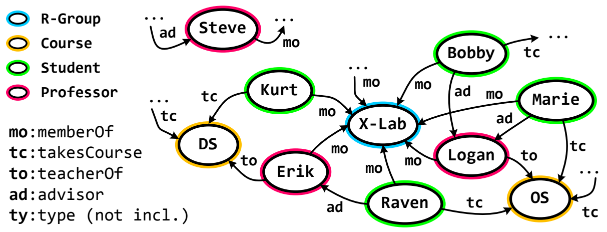


Fig. 1: An example RDF graph.

2 Background

2.1 RDF and SPARQL

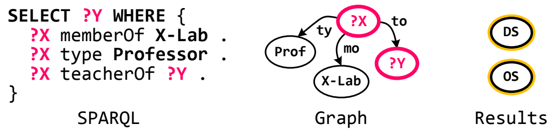
An RDF dataset is a graph (aka RDF graph) composed by triples, where a triple is formed by $\langle subject, predicate, object \rangle$. A triple can be regarded as a directed edge (*predicate*) connecting two vertices (from *subject* to *object*). Thus, an RDF graph can be alternatively viewed as a 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 [3], the entity Steve 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.

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



```
SELECT ?Y WHERE {
  ?X memberOf X-Lab .
  ?X type Professor .
  ?X teacherOf ?Y .
}
```

Fig. 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 X-Lab. The query can also be graphically represented by a query graph, in which vertices represent the subjects and objects of the triple patterns; the black vertices represent constants, and the red vertices represent variables; The edges represent predicates in the required patterns (GP). The query results (?Y, described in RD) include DS and OS.

Difference from graph analytics. Readers might be curious about the relationship between RDF queries and graph analytics [28, 18, 31, 19, 13, 41, 55, 56], especially a recent design [50] used one-sided RDMA to implement message-passing primitives. However, there are several fundamental differences between RDF queries and graph analytics.

First, RDF queries are user-centric; thus minimizing the roundtrip latency is more important than maximizing network throughput. Second, RDF queries only touch a small subset of a graph instead of processing the entire graph, making it not worthwhile to dedicate all resources to run a single query. Third, graph-analytics is usually done in a batch-oriented manner in contrast to concurrently serving multiple RDF queries.

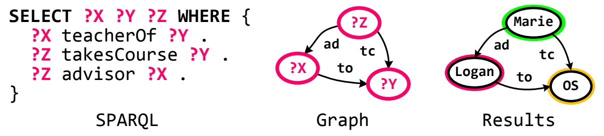
2.2 Existing Solutions

We then 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 X-Lab}, {?X type Professor} and {?X teacherOf ?Y}. For each triple pattern, it generates a temporary query table with bindings by scanning the triple store. In the Join phase, the query tables are joined to produce the final query results.

Some prior work [54] has summarized the 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* [49] can accelerate scan operations, such indexes lead to heavy memory pressure.

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



```
SELECT ?X ?Y ?Z WHERE {
  ?X teacherOf ?Y .
  ?Z takesCourse ?Y .
  ?Z advisor ?X .
}
```

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

Graph store and graph exploration: Instead of joining query tables, Trinity.RDF [49] 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 X-Lab with respect to the data in Figure 1, we find that the possible binding for ?X is only Erik and Logan, 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. 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 (to) the advisor. After exploring all three triple patterns in Q_2 with respect to the data in Figure 1, the non-matching bindings, namely, Logan $\overrightarrow{to}$ OS, OS $\overleftarrow{tc}$ Raven and Raven $\overrightarrow{ad}$ Erik will not be pruned until a final join. Prior work [21, 37] indicates that the final join is a potential bottleneck, especially for queries with cycles and/or large intermediate results.

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 [30, 16, 48], one-sided operations are usually less disruptive than its two-sided counterparts due to no CPU involvement to the target machine. To minimize interference among multiple machines during query processing, we focus on one-sided RDMA operations in this paper. However, it should be straightforward to use two-sided RDMA operations in Wukong as well.

Figure 4(a) shows the throughput (in Kbps) of different communication primitives. RDMA undoubtedly achieves the highest throughput for all payload sizes, while the throughput of TCP/IP over IPoIB (IP over InfiniBand) or 10GbE approaches that of one-sided RDMA

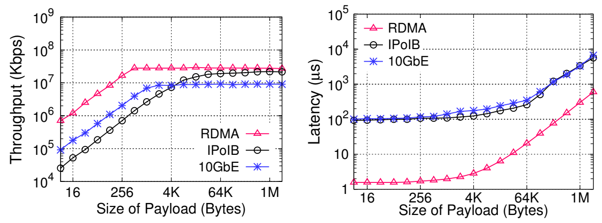


Fig. 4: (a) The throughput and (b) the latency of random reads using one-sided RDMA and TCP/IP with the increase of payload sizes.

with the increase of payload sizes. For payload sizes larger than 4K bytes, the difference is limited to 4 times. In contrast, the gap of roundtrip latencies is always more than an order-of-magnitude, as shown in Figure 4(b). Therefore, it is imperative to leverage one-sided RDMA operations (i.e., READ and WRITE) to boost latency-oriented query processing. Further, an interesting feature is that the latency of RDMA is relatively *insensitive* to payload sizes, because small-sized requests cannot saturate the high-bandwidth network card[5]. For example, the latency only increases slightly (from 1.56μs to 2.25μs) even if the payload size increases 256X (from 8 bytes to 2K bytes).

3 Overview

Setting: Wukong assumes a cluster that is connected with a 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 duplicate edges to make sure each machine contains a self-contained subgraph (e.g., no dangling edges) of the input RDF graph, for better locality. Wukong also creates index 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.

Architecture: An overview of Wukong's architecture is shown in Figure 5. Wukong follows a decentralized model on the server side, where each machine can directly serve clients' requests. Each client[6] 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. Some sophisticated mechanisms like congestion control [57] and load balancing [36] can also be implemented at the proxy, which are beyond the scope of this paper. Moreover, to avoid sending and storing long strings and thus save network bandwidth

[5]Note that this features also applies to other communication primitives (e.g., TCP/IP over IPoIB or 10GbE).

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

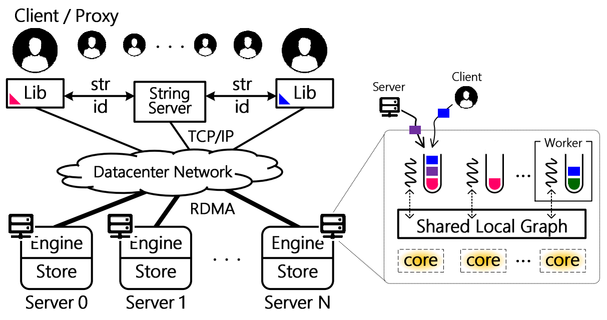


Fig. 5: The architecture overview of Wukong.

and memory consumption, each string is first converted into a unique ID by the string server, similar to prior work [54, 21].

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 from clients or other servers. The graph store layer adopts an RDMA-friendly key/value store over distributed hashtable to support a partitioned global address space. Each machine stores a partition of the RDF graph, which is shared by all of worker 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 vertex. For a query starting with a set of vertices with a specific type or predicate, Wukong then sends the request to all replicas of the corresponding index vertex.

Wukong parses a query into an operator tree, the same as other systems. 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 will be pushed into the task queue to be scheduled and executed asynchronously.

4 Graph-based RDF Data Modeling

This section provides a detailed description of the graph indexing, partitioning and storing strategies employed by Wukong, which are the basis 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 kinds of *index* vertices to assist

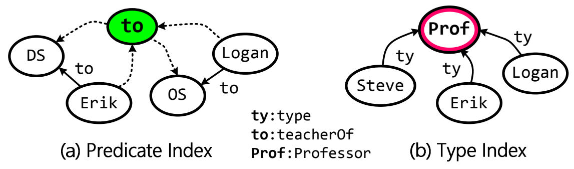


Fig. 6: Two types of index vertex of Wukong.

such queries, as shown in Figure 6. To avoid confusion, we use the *normal* vertex to refer to subjects and objects.

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 subjects and objects labeled with the particular predicate using its in and out edges respectively. The index vertex essentially serves as an inverted index from the predicate to the corresponding subjects or objects. For example, in Figure 6, a predicate index teacherOf (to) links to all normal vertices whose in-edges (DS and OS) or out-edges (Erik and Logan) contain the label to. This corresponds to the PSO and POS indexes in the 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), instead of providing a uniform but useless predicate index type to link all objects and subjects. 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 an RDF graph and also takes into consideration the partitioning and storing of such indexes. 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 [13], Wukong adopts differentiated partitioning algorithms to normal and index vertices. One difference is that unlike PowerLyra, Wukong does not use the degrees to differentiate vertices, because an RDF query only navigates through a vertex and then routes to only a portion of its neighbors. Therefore, unlike graph analytics, a high-degree vertex in skewed

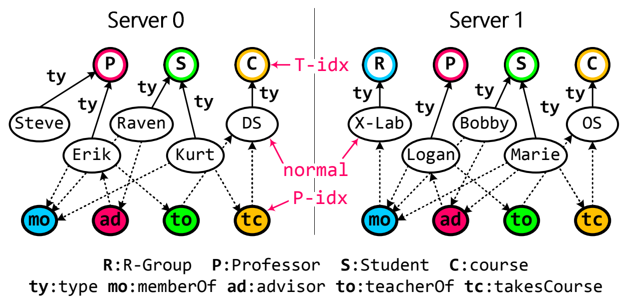


Fig. 7: A hybrid graph partitioning on two servers.

graphs does not necessarily incur significant imbalance for query processing, and it can be handled by fork-join execution appropriately (§ 5.3).

As shown in Figure 7, each normal vertex (e.g., DS) will be randomly assigned (i.e., by hashing the vertex ID) to only one machine with all of its edges (IDs of neighbors). 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 no need to find a predicate index vertex via normal vertices and this can save plenty of memory. Different from a 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 load among each machine.

4.3 RDMA-friendly Predicate-based Store

Similar to Trinity.RDF [54], 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 an 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 information in the key would lead to plenty of unnecessary computation cost and networking traffic. The finer-grained vertex decomposition 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 predicate and type indexes respectively. Figure 8 illustrates part of detailed cases on the sample graph. The key of normal vertex starts from a nonzero vid

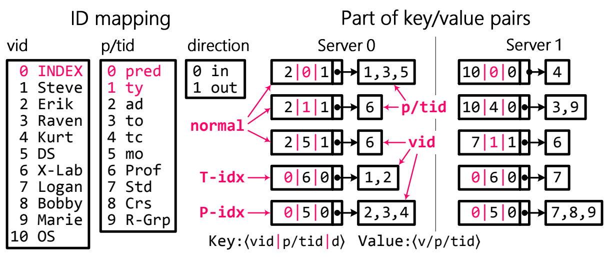


Fig. 8: The design of predicate-based key/value store.

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 Erik is represented as the key ⟨2|0|1⟩, and the value ⟨1,3,5⟩ means type, teacherOf and memberOf. While the type of vertex Erik is represented as the key ⟨2|1|1⟩, and the value ⟨6⟩ means Professor. The key of an index vertex always starts from a zero vid, and linked to a list of local normal vertices. For example, all subjects of the predicate memberOf on Server 0 (Erik, Raven and Kurt) and Server 1 (Logan, Bobby and Marie) are stored with the same key ⟨0|5|0⟩ but on 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 [48] 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. Likewise, other *symmetric* RDMA-friendly stores [16] can also work with Wukong to store RDF graph and support query processing (§5).

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 [54]. Hence, like prior work [54], 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 is a free variable. For the common cases where the predicate is known but the subject/object are free variables, Wukong can leverage the predicate index to begin the graph exploration. Take 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 a cost-based approach with some heuristics.

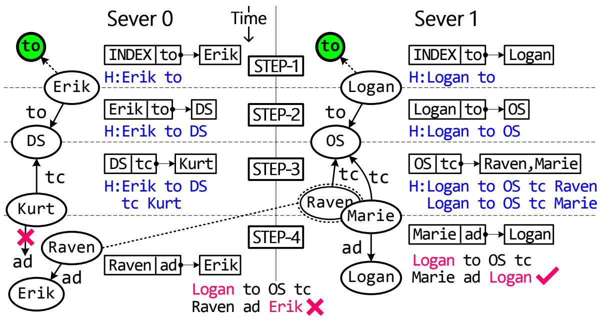


Fig. 9: A sample of execution flow on Wukong. The blue label H: shows the full history.

As shown in Figure 9, Wukong starts exploration from the teacherOf predicate (to). Since Wukong extends the graph with predicate indexes, it can start exploration from the index vertex for teacherOf in each machine in parallel, whose neighbors contain Erik and Logan in each server accordingly. In Step2, Wukong combines Erik and Logan with teacherOf to form the key to get the corresponding courses, which are {Erik $\overrightarrow{to}$ DS} and {Logan $\overrightarrow{to}$ 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 reserved p/tid 0 (pred). The value is the list of predicates associated with the vertex, and then Wukong iterates over 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 Kurt, the related tuples should be filtered out to minimize redundant computation and communication. Further, in Step 4, as Raven's advisor is Erik instead of Logan, the graph exploration path also should be pruned as well.

Prior graph-exploration strategies [54] usually use a one-step pruning approach by leveraging the constraint in the immediately prior step to filter out unnecessary data (e.g., only DS and OS in Step 3). 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 [21, 37] 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. Our experiment on LUBM [3] shows that some query spends more than 90% of execution time on the final join (details in §7.3).

Instead, Wukong adopts a *full-history pruning* approach such that Wukong passes the full exploration history to the next step within or across machines. The main observation is that, the cost of RDMA operations is insensitive to the payload size when it is smaller than a certain size (e.g., 2K bytes). Besides, the steps and variables in an RDF query are usually not many (i.e., less than 10), and each history item only contains subject/object/predicate IDs. Thus there won't be too much information carried even for the final few steps. Consequently, the cost remains low even passing more history information across machines. Further, improving the locality of graph exploration can also avoid additional network traffic from the full-history pruning.

As shown in Figure 9, Wukong passes {Erik $\overrightarrow{to}$}, {Erik $\overrightarrow{to}$ DS} and {Erik $\overrightarrow{to}$ DS $\overleftarrow{tc}$ Kurt} locally on Server 0 in each step; Kurt can be simply pruned without using history information due to no expected predicate (ad). Server 0 can leverage the full history ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven}) from Server 1 to prune Raven as Raven's advisor is not Logan.

As Wukong has the full history during graph exploration, there is no need of a final join to filter out non-matching results. Though 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. For example, the query L1 on LUBM-10240 can benefit from early pruning to save about 96% network traffic (462MB vs. 18MB). Besides, many query histories are passed within a single machine and thus do not cause additional network traffic. In case the full history size is excessively large, Wukong can adaptively fall back to one-step pruning for the sub-query. However, we did not encounter such a case during our evaluation.

5.3 Migrating Execution or Data

During the graph exploration process, there will be different tradeoffs on whether migrating data or execution. Wukong provides *in-place* and *fork-join* executions accordingly. For a query step, if only a few vertices need to be fetched 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 Raven (i.e., Erik) by one RDMA READ, and locally generate ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven $\overrightarrow{ad}$ Erik}).

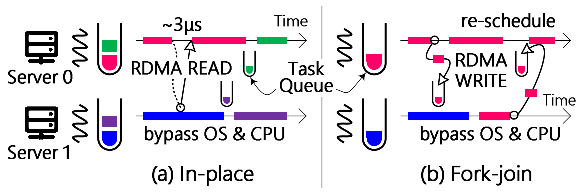


Fig. 10: A sample of (a) in-place and (b) fork-join execution.

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 into 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 ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven}) to Server 0. Server 0 will locally execute the sub-query to generate ({Logan $\overrightarrow{to}$ OS $\overleftarrow{tc}$ Raven $\overrightarrow{ad}$ Erik}). 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. In addition, all of sub-queries will be executed asynchronously without any global barrier and communication among worker threads. Even if two sub-queries access the same vertex, they are still independent due to working on different exploration paths.

Since the cost of RDMA operations is insensitive to the size of the payload, for each query step, Wukong decides on the execution mode at runtime according to the number of RDMA operations ($|N|$) for the next step. for the fork-join mode, $|N|$ is twice the number of servers; for the in-place mode, $|N|$ is equal to the number of required vertices. Each server will decide individually. Wukong simply uses a heuristic 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 oversized payload. Wukong can label such vertices associated with the predicate to force the use of the fork-join mode when partitioning the RDF graph.

5.4 Concurrent Query Processing

Depending on the complexity and selectivity, the latency (i.e., execution time) of a query may vary significantly. For example, the latency differences among seven

```
int next = 1

OBLIGER()
  s = state[(tid+next)%N]
  q = NULL
  s.lock()
  if (s.cur == tid //reentry
   || s.end < now)
    s.cur = tid;
    s.end = now + T
    next++
    q = s.dequeue()
  s.unlock()
  return q

SELF()
  s = state[tid]
  s.lock()
  s.cur = tid
  s.end = now + T
  next = 1
  q = s.dequeue()
  s.unlock()
  return q

NEXT_QUERY()
  if (q = OBLIGER())
    return q
  return SELF()
```

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

queries in LUBM [3] can reach around 3,000X (0.17ms and 516ms for L5 and L1 queries accordingly). Hence, dedicating an entire cluster for a single query, as done in prior approaches [54, 21], 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 inter-query interference while providing good utilization of resources, e.g., a lengthy query should not significantly extend the latency of a fast 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.

The work stealing mechanism [10] is widely used to provide load balance in parallel systems, which allows tasks can be stolen from any queue of worker threads. However, the traditional algorithm is inefficient as the stolen tasks in Wukong are mostly sub-millisecond latency queries. Further, the unrestricted stealing among all workers may incur large overhead due to high contention.

To this end, Wukong uses a *worker-obliger* work stealing algorithm for multiple workers on 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 a timeout (i.e., `s.end < now`). If so, that worker might be handling a lengthy query and thus its following up queries may be delayed. In this case, this obliging worker steals one query from that worker's queue to process. After obliging its neighboring workers (until seeing a non-busy one), the worker

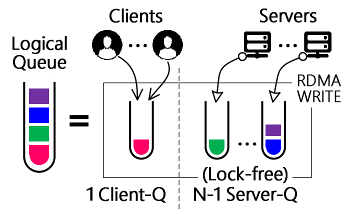


Fig. 12: The logical task queue in Wukong.

will then continue to handle its own queries by dequeuing from its own worker queue.

Note that, when all workers can handle their queries within a time 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 in usual. It could be possible that an obliger get stucked in handling a lengthy query for others; in this case, another worker may oblige this worker 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 X-Lab}) 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 same query to all servers such that the query can be dis-

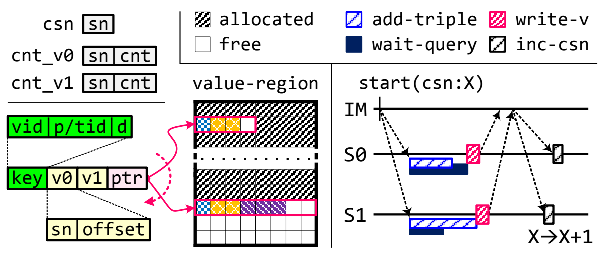


Fig. 13: The extension of graph store and the execution flow of injection for evolving RDF graphs.

tributed 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 the client. 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 the MT threshold of the server.

Evolving graph While most prior RDF stores only support read-only queries, Wukong is also built with preliminary support to incrementally update the graph with concurrent queries. New triples will be periodically ingested to the RDF store, and all queries will run a consistent snapshot. Figure 13 illustrates three extensions to Wukong to support incremental update.

RDF Store. To support the dynamic increase of value, Wukong provides a buddy memory allocator. When the value space is full, the allocator will find a free value with double capacity, copy all data of the old value to the new one, and replace the pointer of the key using an atomic instruction. Further, to provide a consistent snapshot to above queries, each key should be extended with two versions (v0 and v1) that consist of its snapshot number and the offset within its value. The left part of Figure 13 illustrate the extension of RDF store.

Query processing. On each machine, there are two global reference counters (cnt_v0 and cnt_v1) to record

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

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

the number of outstanding queries on two latest snapshots, and a current snapshot number (csn). Each query will first read the current snapshot number, and actively increase and decrease the corresponding counter before and after execution. The snapshot number of a query will be used to fetch a consistent version of all values and be inherited by all of its sub-queries.

RDF data injection. The added RDF triples in the new graph will be locally injected into all servers, which is coordinated by a single injection master (IM). Wukong performs the injection by executing the following steps. First, all triples are added in the background and remain invisible to concurrent queries. Meanwhile, all outstanding queries on the older snapshot (between v0 and v1) should be completed in advance. After they are done, each server will safely overwrite the older version within the keys by the new one and notify IM. When all servers are ready, IM will finally ask all servers to finish the injection of the new snapshot by atomically increasing the current snapshot number (csn) and the older global counter (between cnt_v0 and cnt_v1). The right part of Figure 13 shows the execution flow of the injection of the snapshot X+1 on two servers (S0 and S1).

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 64GB of DRAM. Each machine is equipped with two ConnectX-3 MCX353A 56Gbps InfiniBand NICs via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps IB Switch, and an Intel X520 10GbE NIC connected to a Force10 S4810P 10GbE 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 [48, 17, 47, 46]. 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 Leigh University Benchmark (LUBM) [3] and the Waterloo SPARQL Diversity Test Suite (WSDTS) [5]. For LUBM, we generate 5 datasets with different sizes using the generator v1.7 in NT format. For queries, we

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

LUBM 2560	Wukong	TriAD	TriAD-SG (50K)	RDF-3X (mem)	BitMat (mem)
L1	752	621	3,315	2.3E5	abort
L2	120	149	221	4,494	36,256
L3	306	316	3,101	3,675	752
L4	0.19	3.38	3.34	2.2	55,451
L5	0.11	2.34	1.36	1.0	52
L6	0.56	20.7	6.06	37.5	487
L7	671	2,176	2,753	9,927	19,323
Geo. M	15.7	72.3	108	441	–

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	78	512	695	6,081	4.4E6
L3	203	1,252	1,225	8,735	12.9E6
L4	0.41	3.4	3.9	5	10.6E6
L5	0.17	3.1	4.5	4	4.2E6
L6	0.89	63	4.6	9	8.7E6
L7	464	10,055	11,572	31,214	12.0E6
Geo. M	16	190	141	450	9.1E6

use the benchmark queries published in Atre et al. [9], which were widely used by many distributed RDF systems [21, 54, 27]. WSDTS publishes a total of 20 queries in four categories. The real-life datasets are the DBpedia's SPARQL Benchmark (DBPSB) [1] and YAGO2 [6, 22]. 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+ [37].

Comparing targets: We compare the query performance of Wukong against several state-of-the-art systems. 1) centralized systems: RDF-3X [33] and BitMat [9]; 2) distributed systems: TriAD [21], Trinity.RDF [54] and SHARD [40]. Since Trinity.RDF is not publicly available and TriAD reported superior performance over it, we only directly compare the results published in their paper [54] with the same workload. Except Wukong, all systems run over InfiniBand using IPoIB. We also enable string server for all systems to save memory consumption, reduce network bandwidth, and boost string matching.

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 run Wukong and TriAD on a single machine and report the in-memory performance of RDF-3X and BitMat. As shown in Table 2[7], Wukong has significantly outperformed RDF-3X and BitMat by several orders of magnitude, due to fast graph exploration for simple queries and efficient multi-threading for complex queries. Note that L3 has an empty final result even with huge intermediate results and thus there is no significant performance difference between Wukong and BitMat. TriAD also enables multi-threading and provides similar performance compared to Wukong for large (non-selective) queries. However, for small (selective) queries, Wukong is still at least an order-of-magnitude faster than TriAD due to the fast graph exploration, even without the optimizations aiming at distributed environment.

We further compare Wukong with distributed systems with multi-threading enabled using LUBM-10240 in Table 3. For small queries (L4, L5 and L6), Wukong outperforms TriAD by up to 70.6X (from 8.4X) mainly due to the in-place execution with one-sided RDMA READ. For large queries (L1, L2, L3 and L7), Wukong still outperforms TriAD by up to 21.7X (from 4.1X), thanks to the fast graph exploration with indexing vertex 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 9.0X (ranging from 2.8X to 26.6X). Compared to Trinity.RDF, which also uses graph-exploration strategy, the improvement of Wukong is at least one order of magnitude (from 10.1X to 78.0X), 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 machines. 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.

Table 4: The query latency (msec) of Wukong on evolving LUBM with 1 million triples/second ingestion rate.

LUBM-10240	L1	L2	L3	L4	L5	L6	L7
Wukong	587	87	222	0.43	0.18	0.95	516
Overhead (%)	12.0	10.3	8.6	4.7	5.6	6.3	10.1

Evolving RDF Graphs: To investigate the performance of Wukong on a continually growing graph, we ingest triples to the LUBM-10240 with the rate of 1 million triples per second on our 6-node cluster, while simultaneously handling queries. Currently, Wukong adopts a queries-friendly design, which minimizes the impact on query processing. The main overhead is from the versioning read. As shown in Table 4, the performance overhead of latency is only about 10.3% and 5.5% for large (L1, L2, L3 and L7) and small (L4, L5 and L6) queries respectively, depending on the number of data accessing.

7.3 Factor Analysis of Improvement

To study the impact of each design decision and how they affect the query performance, we iteratively enable each optimization and collect the query latency using the LUBM-10240 dataset, as shown in Table 5:

[7]LUBM-2560 is used due to limited main memory of a single machine, where the average (geometric mean) latency of Wukong on 6 machines is 7.5 msec.

Table 5: The contribution of optimizations to query latency (msec) of Wukong. Optimizations are cumulative.

LUBM 10240	**BASE**	**+RDMA**	**+FHP**	**+IDX**	**+PBS**	**+DYN**
L1	9,766	9,705	888	853	814	516
L2	2,272	2,161	1,559	84	79	78
L3	421	404	404	205	203	203
L4	1.49	0.79	0.78	0.78	0.56	0.41
L5	1.00	0.39	0.39	0.39	0.31	0.17
L6	3.84	1.40	1.37	1.37	1.17	0.89
L7	2,176	2,041	657	494	466	464
Geo. M	102.3	69.1	39.6	22.6	19.9	15.7

- **BASE**: leverages graph-exploration strategy with one-step pruning. The communication adopts message passing over TCP/IP.
- **+RDMA**: uses one-sided RDMA operations to improve the communication.
- **+FHP**: enables full-history pruning (§5.1 and 5.2).
- **+IDX**: adds two types of index vertex (§4.1) and differentiated graph partitioning (§4.2).
- **+PBS**: leverages predicate-based finer-grained vertex decomposition (§4.3).
- **+DYN**: supports in-place execution and dynamically switches between data migration and execution distribution (§5.3).

Overall, all optimizations (**+DYN**) improves the average (geometric mean) latency by 6.5X over the basic version (**BASE**). The basic version already outperforms TriAD for small queries by leveraging graph exploration, while having inferior performance for large queries due to the overhead of the (expensive) final join operations. Note that Wukong can detect the empty final result of L3 in early steps and thus avoid the final join.

Leveraging RDMA for communication (**+RDMA**) improves the baseline performance slightly (ranging from 1% to 7%) for large queries and about twice (ranging from 1.9X to 2.7X) for small queries, depending on the proportion of communication cost. By skipping the costly final join, enabling full-history pruning (**+FHP**) notably accelerates the non-selective queries. The index vertex with differentiated partitioning (**+IDX**) can improve the parallelism and reduce network traffic for large queries launching from a set of entities (subject/object) with a certain predicate or type, especially for L2. L2 collects a large number of entities (i.e., Courses) on each machine, which can be avoided by decentralizing index vertex. Using predicate-based graph store (**+PBS**) further notably reduces the latency of small queries (ranging from 1.2X to 1.4X), due to finer-grained vertex decomposition by predicates. Finally, the *in-place* execution can bypass remote CPU and OS and avoid the overhead of task scheduling by leveraging one-sided RDMA READ to fetch remote data. Therefore, the optimization (**+DYN**) improves the performance by up to 1.8X.

Table 6: A comparison of query latency (msec) with different execution modes.

LUBM 10240	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**
In-place	21,859	80	204	0.42	0.17	2.43	12,068
Fork-join	813	79	203	0.63	0.47	1.27	466
Dynamic	516	78	203	0.41	0.17	0.89	464

To further study the benefit of dynamic execution mode switching in each step, we configure Wukong with a fixed mechanism (i.e. *in-place* or *fork-join*). As shown in Table 6, in-place mode is beneficial for L4 and L5, while fork-join execution is beneficial for 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 modes for achieving optimal performance. Wukong can always choose the right mode in runtime and thus outperform in-place and fork-join mode alone by up to 42.3X and 2.8X. Note that the poor performance of L1 and L7 with in-place mode is caused by massive small-sized RDMA READs.

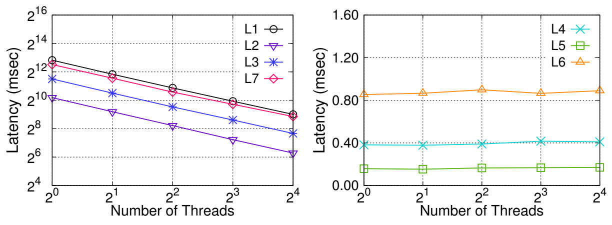


Fig. 14: The latency of queries in group (I) and (II) with the increase of threads on LUBM-10240.

7.4 Scalability

We evaluate the scalability of Wukong in three aspects by scaling the number of threads, the number of machines, and the size of dataset accordingly. We categorize seven queries on LUBM dataset into two groups according to the sizes of intermediate and final results as done in prior work [54]. Group (I): L1, L2, L3, and L7; the results of such queries increase with the growing of dataset. Group (II): L4, L5, and 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 14 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 evaluated 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

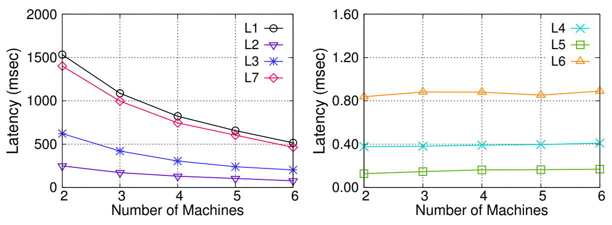


Fig. 15: The latency of queries in group (I) and (II) with the increase of machines on LUBM-10240.

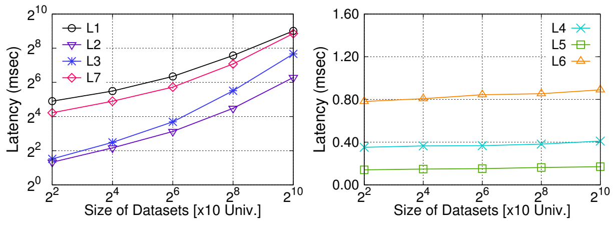


Fig. 16: The latency of queries in group (I) and (II) with the increase of LUBM datasets (40-10240).

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 evaluated 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 16, 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.

Wukong is a good practicer of the COST metric [29], which pursues scalable parallelism for large queries and efficient use of resources for small queries.

7.5 Throughput of Mixed Workloads

Unlike prior RDF stores [54, 21] 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 a mixed workload 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[8], a single client will send queries one by one since it only can handle one query at a time.

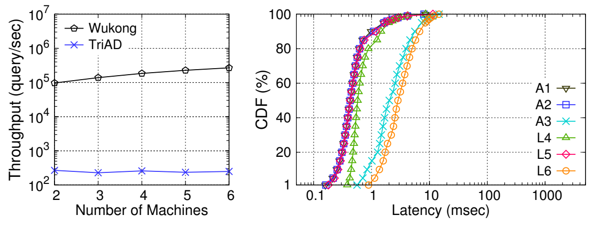


Fig. 17: (a) The throughput of a mixture of queries with the increase of machines, and (b) the CDF of latency for 6 classes of queries on 6 machines.

We first use a mixture workload consisting of 6 classes of queries[9], all of which disable multi-threading. The 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 17, Wukong achieves a peak throughput of 269K queries/second on 6 machines (97K queries/second on 2 machines), which is at least two orders of magnitude higher than TriAD (from 278X to 740X). Under the peak throughput, the geometric mean of 50th (median) and 99th percentile latency is just 0.80 and 5.90 milliseconds respectively.

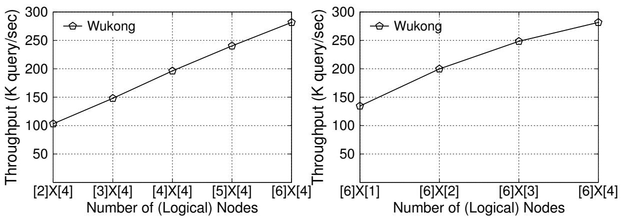


Fig. 18: The throughput of a mixture of queries with the increase of logical nodes. The tick labels of x-axis are the configuration, and the symbol of `[m]X[n]` corresponds with `#machines` and `#nodes/machine`.

Scalability with logical nodes: To overcome the restriction of cluster size, we emulate a large cluster by scaling the logical nodes on each machine and evaluate the throughput of Wukong along with the increase of logical nodes. Each logical node has 4 worker threads and the interaction between logical nodes still uses one-sided RDMA operations even on the same machine. As shown in Figure 18, Wukong scales out to 24 nodes by both the number of machines and the number of nodes per machine; the throughput reaches 282K queries per second.

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

[8]We are not aware of open-sourced RDF systems supporting concurrent query processing. On the other hand, existing graph databases or graph-analytics systems have even worse performance compared to TriAD due to the lack of RDF and SPARQL supporting.

[9]The templates of 6 classes of queries are based on group (II) queries (L4, L5, and L6) and three additional queries (A1, A2, and A3) from the official LUBM website (#1, #3, and #5).

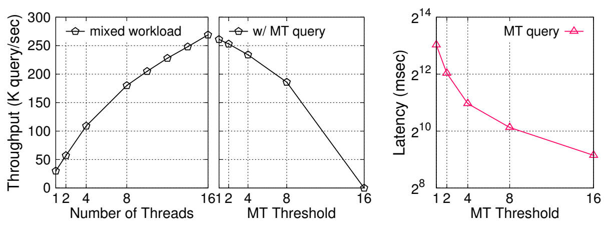


Fig. 19: (a) The throughput of a mixture of queries with the increase of threads, (b) the throughput with multi-threaded (MT) queries under various MT thresholds, and (c) the average latency of multi-threaded (MT) queries.

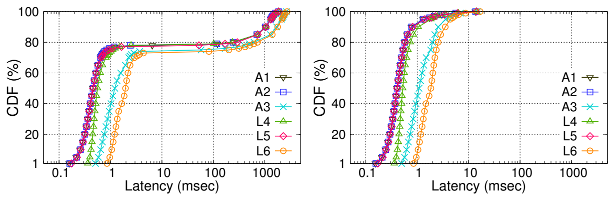


Fig. 20: The CDF of latency for 6 classes of queries on 6 machines (a) w/o and (b) w/ worker-obliger mechanism. Each server uses fixed 8 threads (threshold=4).

queries (i.e., L1) and configure Wukong with different MT thresholds. As shown in Figure 19(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 the threshold 8, Wukong can still perform 186K queries/second and the average latency of MT query is about 1,118 msec.

Worker-obliger mechanism: The MT query will also influence the latency of other small queries in the waiting queues. Figure 20(a) show the CDF graph 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. With the worker-obliger mechanism, as shown in Figure 20(b), Wukong can notably reduce the query latency while preserving the throughput.

LUBM 2560	MEM (GB)
Wukong	18.5
BASE	12.3
TriAD	30.7
RDF-3X	24.2
BitMat	27.4

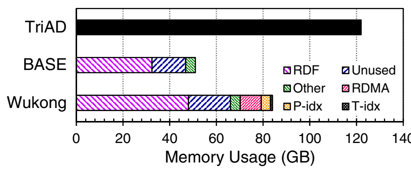


Fig. 21: A comparison of memory usage and breakdown on various systems for (a) LUBM-2560 and (b) LUBM-10240. The storage size is 6.2GB and 25GB respectively.

7.6 Memory Consumption

Readers might be interested in how the memory consumption of Wukong compares to other state-of-the-art systems. Triple stores, including TriAD, RDF-3X, and BitMat, rely on redundant *six* primary SPO permutation indexes [49] to accelerate querying, which, however, lead to high memory pressure. In contrast, managing RDF data in native graph form is much space-efficient, which only doubles the triples in RDF for subjects and objects. Figure 21(a) compares the memory usage of various systems for LUBM-2560 on a single machine. All triple stores consume much more memory compared to Wukong, especially for its basic version (i.e., BASE).

Figure 21(b) further shows a breakdown of memory usage in Wukong for LUBM-10240 on the 6-node cluster. Compared to the base version, Wukong adds about 3.9GB and 0.9GB memory for predicate index (P-idx) and type index (T-idx), as well as additional 15.5GB memory for RDF to support predicate-based store. Furthermore, 9.0GB memory (1.5GB per machine) is reserved for one-sided RDMA operations. Note that the underlying key/value store of Wukong is a hashtable with less than 75% occupancy, because Wukong is currently not well tuned for high space-efficiency.

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.

Table 7: 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	0.9	3.6	10.3

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 7 shows the geometric mean of latency for various query classes. Wukong always outperforms TriAD by up to 58.2X (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 8: The latency (msec) of queries on DBPSB

DBPSB	D1	D2	D3	D4	D5	Geo. M
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 8 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.

YAGO2: Table 9 compares the performance of TriAD and Wukong on a large real-life dataset YAGO2. For the

Table 9: The latency (msec) of queries on YAGO2

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

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 [33, 34, 8, 49, 42, 11]. Since join is expensive and a key step for query processing in such triple stores, they perform various query optimizations including heuristic optimizations [33], join-ordering exploration [33], join-ahead pruning [34], and query caching [39]. Specially, TriAD [21] is a recent distributed in-memory RDF engine that leverages join-ahead pruning and graph summarization with asynchronous message passing for parallelization. SHAPE [27] is a distributed engine upon RDF-3X by statically replicating and prefetching data. As shown in prior work [54], graph exploration avoids many redundant immediate results generated during expensive join operations and thus typically delivers better performance. A recent study, SQLGraph [45], leverages a relational store to store RDF data but processes RDF queries as a graph store. Yet, it focuses on query rewriting and schema refinement to support ACID-style transactions and thus has different objectives from Wukong.

RDF query over graph store: There is an increasing interest in using native graph model to store and query RDF data [9, 53, 58, 52, 54]. BitMat [9], gStore [58] and TripleBit [53] are centralized graph stores with sophisticated indexes to improve query performance. Sedge [52] 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 [54], 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 [38, 37, 40, 43], e.g., H_2RDF [38, 37] and SHARD [40]. PigSPARQL [43] maps SPARQL operations into PigLatin [35] 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 [21, 54].

Graph databases and query systems: Neo4j [2] and HyperGraphDB [24] focus on supporting online transaction processing (OLTP) on graph data; however they are not distributed and cannot support web-scale graphs partitioned over multiple machines. Titan [4] instead supports distributed graph traversals over multiple machines, which, however, does not support SPARQL queries. Facebook's TAO [12] provides a simple API and data model to store and query geographically distributed data. Unicorn [15] further leverages TAO as the storage layer to support searching over the social data. To our knowledge, none of the above systems exploit RDMA as well as the optimization techniques in Wukong to boost query latency and throughput.

RDMA-centric stores: The low latency and high throughput of RDMA-based networking stimulate much work on RDMA-centric key/value stores [30, 25], OLTP platforms [48, 17, 14] and graph analytics engines [50, 23]. Specifically, GraM [50] 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 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 graph-analytics task at a time. Recently, Kalia et al. [26] provide several of RDMA design space for system designers.

9 Conclusion

This paper describes Wukong, a distributed in-memory RDF store that leverages RDMA-based graph exploration to support fast and concurrent SPARQL queries. Wukong significantly outperforms state-of-the-art systems and can process a mixture of small and large queries at 269K queries/second on a 6-node RDMA-capable cluster. Currently, we only consider the SPARQL query over timeless RDF datasets; our future work may extend Wukong to support RDF stream processing (RSP)[10].

10 Acknowledgments

We sincerely thank our shepherd Dushyanth Narayanan and the anonymous reviewers for their insightful suggestions, as well as Yunhao Zhang for sharing his experience to support evolving RDF graphs. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61402284, 61572314) and the Zhangjiang Hi-Tech program (No. 201501-YP-B108-012).

[10]https://www.w3.org/community/rsp/

References

[1] DBpedias SPARQL Benchmark. http://aksw.org/Projects/DBPSB.

[2] Neo4j Graph Database. http://neo4j.org/.

[3] SWAT Projects - the Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[4] Titan: Distributed Graph Database. http://titan.thinkaurelius.com/.

[5] Waterloo SPARQL Diversity Test Suite (WSDTS). https://cs.uwaterloo.ca/~galuc/wsdts/.

[6] YAGO: A High-Quality Knowledge Base. http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago.

[7] Bio2RDF: Linked Data for the Life Science. http://bio2rdf.org/, 2014.

[8] 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.

[9] 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.

[10] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[11] 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.

[12] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., ET AL. Tao: Facebooks distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 49–60.

[13] 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.

[14] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[15] CURTISS, M., BECKER, I., BOSMAN, T., DOROSHENKO, S., GRIJINCU, L., JACKSON, T., KUNNATUR, S., LASSEN, S., PRONIN, P., SANKAR, S., SHEN, G., WOSS, G., YANG, C., AND ZHANG, N. Unicorn: A system for searching the social graph. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1150–1161.

[16] 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.

[17] 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.

[18] GONZALEZ, J., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012).

[19] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *OSDI* (2014).

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

[21] 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.

[22] 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.

[23] 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.

[24] IORDANOV, B. Hypergraphdb: a generalized graph database. In *Web-Age information management*. Springer, 2010, pp. 25–36.

[25] 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.

[26] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (2016), USENIX ATC'16.

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

[28] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[29] MCSHERRY, F., ISARD, M., AND MURRAY, D. Scalability! But at what COST? In *HotOS '15* (2015).

[30] 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.

[31] MURRAY, D., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *SOSP* (2013).

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

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

[34] 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.

[35] 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.

[36] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 69–84.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 410–424.

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

[43] 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.

[44] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD* (2013).

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *OSDI* (2016).

[56] ZHU, X., CHEN, W., ZHENG, W., AND XIAOSONG, M. Gemini: A computation-centric distributed graph processing system. In *OSDI* (2016).

[57] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 523–536.

[58] 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.