Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration

Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen, ***Shanghai Jiao Tong University***

https://www.usenix.org/conference/atc18/presentation/wang-siyuan

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration

Siyuan Wang, Chang Lou, Rong Chen, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Contacts: {rongchen, haibochen}@sjtu.edu.cn

ABSTRACT

RDF graph has been increasingly used to store and represent information shared over the Web, including social graphs and knowledge bases. With the increasing scale of RDF graphs and the concurrency level of SPARQL queries, current RDF systems are confronted with inefficient concurrent query processing on massive data parallelism, which usually leads to suboptimal response time (latency) as well as throughput.

In this paper, we present Wukong+G, the first graph-based distributed RDF query processing system that efficiently exploits the hybrid parallelism of CPU and GPU. Wukong+G is made fast and concurrent with three key designs. First, Wukong+G utilizes GPU to tame random memory accesses in graph exploration by efficiently mapping data between CPU and GPU for latency hiding, including a set of techniques like query-aware prefetching, pattern-aware pipelining and fine-grained swapping. Second, Wukong+G scales up by introducing a GPU-friendly RDF store to support RDF graphs exceeding GPU memory size, by using techniques like predicate-based grouping, pairwise caching and look-ahead replacing to narrow the gap between host and device memory scale. Third, Wukong+G scales out through a communication layer that decouples the transferring process for query metadata and intermediate results, and leverages both native and GPUDirect RDMA to enable efficient communication on a CPU/GPU cluster.

We have implemented Wukong+G by extending a state-of-the-art distributed RDF store (i.e., Wukong) with distributed GPU support. Evaluation on a 5-node CPU/GPU cluster (10 GPU cards) with RDMA-capable network shows that Wukong+G outperforms Wukong by 2.3X-9.0X in the single heavy query latency and improves latency and throughput by more than one order of magnitude when facing hybrid workloads.

1 INTRODUCTION

Resource Description Framework (RDF) is a standard data model for the Semantic Web, recommended by W3C [5]. RDF describes linked data as a set of triples forming a highly connected graph, which powers information retrievable through the query language SPARQL. RDF and SPARQL have been widely used in Google's knowledge graph [22] and many public knowledge bases, such as DBpedia [1], PubChemRDF [38], Wikidata [8], Probase [59], and Bio2RDF [10].

The drastically increasing scale of RDF graphs has posed a grand challenge to fast and concurrent queries over large RDF datasets [17]. Currently, there have been a number of systems built upon relational databases, including both centralized [40, 12, 58] and distributed [48, 44, 23] designs. On the other hand, Trinity.RDF [62] uses graph exploration to reduce the costly join operations in intermediate steps but still requires a final join operation. To further accelerate distributed query processing, Wukong [51] leverages RDMA-based graph exploration to support massively concurrency queries with low latency requirement and adopts full-history pruning to avoid the final join operation.

Essentially, many RDF queries have embarrassing parallelism, especially for *heavy* queries, which usually touch a large portion of the RDF graph on an excessive amount of paths using graph exploration. This poses a significant challenge even for multicore CPUs to handle them efficiently, which usually causes lengthy execution time. For example, the latency differences among seven queries in LUBM [7] is more than 3,000X (0.13ms and 390ms for Q5 and Q7 accordingly). This may cause one heavy query block all other queries, substantially extending the latency of other queries and dramatically impairing the throughput of processing concurrent queries [51]. This problem has also gained increased attention [45].

In this paper, we present Wukong+G[1] with a novel design that exploits a distributed heterogeneous CPU/GPU cluster to accelerate heterogeneous RDF queries based on distributed graph exploration. Unlike CPUs pursuing the minimized execution time for single instructions, GPUs are designed to provide high computational throughput for massive simple control-flow operations with little or no control dependency. Such features expose a design space to distribute hybrid workloads by offloading heavy queries to GPUs. Nevertheless, different from many traditional GPU workloads, RDF graph queries are memory-intensive instead of compute-intensive: there are limited arithmetic operations and most of the processing time is spent on random memory accesses. This unique feature implies that the key of performance optimizations in Wukong+G is on *smart*

[1]The source code and a brief instruction of Wukong+G are available at http://ipads.se.sjtu.edu.cn/projects/wukong.

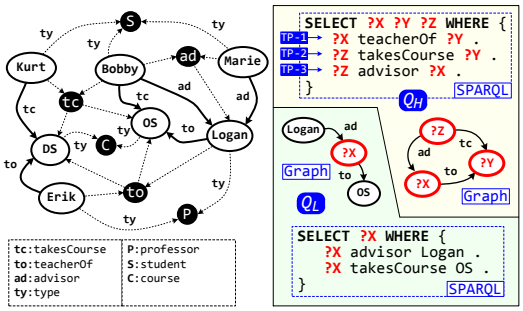

Fig. 1: A sample of RDF data and two SPARQL queries (Q_H and Q_L). White circles indicate the normal vertices (*subjects* and *objects*); dark circles indicate the (*type* and *predicate*) index vertices. Q_H is a *heavy* query, and Q_L is a *light* query.

memory usage rather than improving the computation algorithm. Wukong+G is made fast and concurrent with the following key designs:

GPU-based query execution (§4.1). To achieve the best performance for massive random accesses demanded by heavy queries, Wukong+G leverages the many-core feature and latency hiding ability of GPUs. Besides making use of hardware advantages, Wukong+G surmounts the limitations of GPU memory size and PCIe (PCI Express) bandwidth by adopting *query-aware prefetching* to mitigate the constraints on graph size, *pattern-aware pipelining* to hide data movement cost, and *fine-grained swapping* to minimize data transfer size.

GPU-friendly RDF store (§4.2). To support desired CPU/GPU co-execution pattern while still enjoying the fast graph exploration, Wukong+G follows a distributed in-memory key/value store and proposes a *predicate-based grouping* to aggregate keys and values with the same predicate individually. Wukong+G further smartly manages GPU memory as a cache of RDF store by supporting *pairwise caching* and *look-ahead replacing*.

Heterogeneous RDMA communication (§4.3). To preserve better communication efficiency in a heterogeneous environment, Wukong+G decouples the transferring process of query metadata and intermediate results for SPARQL queries. Wukong+G uses native RDMA to send metadata like query plan and current step among CPUs, and uses GPUDirect RDMA to send current intermediate results (history table) directly among GPUs. This preserves the performance boost brought by GPUs from potential expensive CPU/GPU data transfer cost.

We have implemented Wukong+G by extending Wukong [51], a state-of-the-art distributed RDF query system to support heterogeneous CPU/GPU processing. To confirm the performance benefit of Wukong+G, we have conducted a set of evaluations on a 5-node CPU/GPU cluster (10 GPU cards) with RDMA-capable network. The experimental results using the LUBM [7] benchmark show that Wukong+G outperforms Wukong by 2.3X-9.0X in the single heavy query latency and improves latency and throughput by more than one order of magnitude when facing hybrid workloads.

2 Background and Motivation

2.1 RDF and SPARQL

An RDF dataset is composed by triples, in the form of $\langle subject, predicate, object \rangle$. To construct a graph (aka RDF graph), each triple can be regarded as a directed edge (*predicate*) connecting two vertices (from *subject* to *object*). In Fig. 1, a simplified sample RDF graph of LUBM dataset [7] includes two professors (Logan and Erik), three students (Marie, Bobby, and Kurt), and two courses (OS and DS).[2] There are also three predicates (teacherOf (to), advisor (ad) and takeCourse (tc)) to link them. Two types of indexes, *predicate* and *type*, are added to accelerate query processing on RDF graph [51].

SPARQL, a W3C recommendation, is a standard query language developed for RDF graphs, which defines queries regarding graph patterns (GP). The principal part of SPARQL queries is as follows:

Q := SELECT RD WHERE GP

where (RD) is the *result description* and GP consists of *triple patterns* (TP). The triple pattern looks like a normal triple except that any *constant* can be replaced by a *variable* (e.g., ?X) to match a subgraph. The result description RD contains a subset of variables in the triple patterns (TP) to define the query results. For example, the query Q_H in Fig. 1 asks for professors (?X), courses (?Y) and students (?Z) such that the professor advises (ad) the student who also takes a course (tc) taught by (to) the professor. After exploring all three TPs in Q_H on the sample graph in Fig. 1, the exact match of RD (?X, ?Y and ?Z) is only a binding of Logan, OS, and Bobby.

Query processing on CPU. There are two representative approaches adopted in state-of-the-art RDF systems, (relational) triple join [40, 58, 12, 23] and graph exploration [62, 51]. A recent study [51] found that graph exploration with full-history pruning can provide low latency and high throughput for concurrent query processing. Therefore, we illustrate this approach to demonstrating the query processing on CPU with the sample RDF graph and SPARQL query (Q_H) in Fig. 1.

As shown in Fig. 2, all triple patterns of the query (Q_H) will be iterated in sequence (❶) to generate the results (*history table*) by exploring the graph, which is stored in an in-memory key/value store. According to the variable (?Y) of the current triple pattern (TP-2), each row of a certain column in the history table (❷) will be combined with the constant (takesCourse) of the triple pattern as

[2] Since the special predicate type (ty) is used to group a set of entities, we follow Wukong [51] to treat every type (e.g., professor (P)) as an index, the same as predicates (e.g., advisor (ad)).

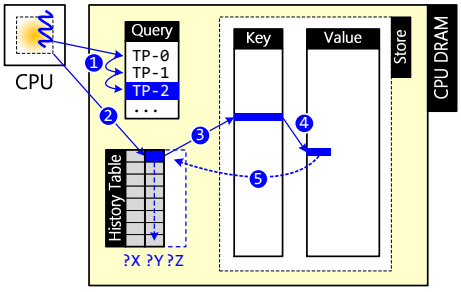

Fig. 2: The execution flow of query processing on CPU.

the key (❸) to retrieve the value (❹). The value will be appended to a new column (?Z) of the history table (❺). Note that an extra triple pattern (TP-0) from an index vertex (teacherOf) will be used to collect all start vertices satisfying a variable (?X) in TP-1.

Full-history pruning. Since processing RDF query by graph exploration needs to traverse the RDF graph, it is crucial to prune infeasible paths for better performance. There are basically two approaches: partial-history pruning [62], by inheriting partial history information (intermediate results) from previous steps of traversing to prune the following traversal paths; and full-history pruning [51], by passing the history information of all previous traversal steps for pruning. Wukong has exploited full-history pruning to prune unnecessary intermediate results precisely and *make all traversal paths completely independent*. Thanks to the fast RDMA-capable network as well as the relative cost-insensitivity of one-sided RDMA operations regarding payload size, full-history pruning is very effective and efficient to handle *concurrent* queries.

Workload heterogeneity. Prior work [62, 23, 51] has observed that there are two distinct types of SPARQL queries: *light* and *heavy*. Light queries (e.g., Q_L in Fig. 1) usually start from a (constant) normal vertex and only explore *a few* paths regardless of the dataset size. In contrast, heavy queries (e.g., Q_H in Fig. 1) usually start from an (type or predicate) index vertex and explore *massive amounts* of paths, which increases along with the growth of dataset size. The top of Fig. 3 demonstrates the number of paths explored by two typical queries (Q5 and Q7) on LUBM-10240 (10 vs. 16,000,000).

The *heterogeneity* in queries can result in tremendous latency differences on state-of-the-art RDF stores [51], even reaching more than 3,000X (0.13ms and 390ms for Q5 and Q7 on LUBM-10240 accordingly).[3] Therefore, the multi-threading mechanism is widely used by prior work [23, 62, 51] to improve the performance of heavy queries. However, such approach is intrinsically restricted by the limited computation resource of CPU. Currently, the maximum number of cores in a commercial CPU processor is usually less than 16. Moreover, the lengthy queries will significantly extend the latency of light queries and impair the throughput of processing concurrent queries. Some CPU systems like Oracle PGX [4] try to address this issue by adopting priority mechanism. However, with no variation of computing power, the sacrifice of user experience for one type of queries is unavoidable.

[3]Detailed experimental setup and results can be found in §6.

2.2 Hardware Trends

Hardware heterogeneity. With the prevalence of computational workloads (e.g., machine learning and data mining applications), it is now not uncommon to see server-class machines equipped with GPUs in the modern datacenter. As a landmark difference compared to CPU, the number of GPU cores (threads) can easily exceed two thousand, which far exceeds existing multicore CPU processors. As shown in Fig. 3, in a typical *heterogeneous* (CPU/GPU) machine, CPU and GPU have their private memory (DRAM) connected by PCIe with limited bandwidth (10GB/s). Compared to host memory (CPU DRAM), device memory (GPU DRAM) has much higher bandwidth (288GB/s vs. 68GB/s) but less capacity (12GB vs. 128GB). Generally, GPU is optimized for performing massive, simple and independent operations with intensive accesses on a relatively small memory footprint.

Fast communication: GPUDirect with RDMA. GPUDirect is a family of technologies that is continuously developed by NVIDIA [3]. Currently, it can support various efficient communications, including inter-node, intra-node, and inter-GPU. RDMA (Remote Direct Memory Access) is a networking feature to directly access the memory of a remote machine, which can bypass remote CPU and operating system, and avoid redundant memory copy. Hence, it has unique features like high speed, low latency and low CPU overhead. GPUDirect RDMA has been introduced in NVIDIA Kepler-class GPUs, like Tesla and Quadro series. This technique enables direct data transfer between GPUs by InfiniBand NICs as the name suggests [2].

2.3 Opportunities

Though prior work (e.g., Wukong [51]) has successfully demonstrated the low latency and high throughput of running light queries solely by leveraging graph exploration with full-history pruning, it is still incompetent to handle heavy queries efficiently. This leads to suboptimal performance when facing hybrid workloads comprising both light and heavy queries.

This problem is not due to the design and implementation of existing state-of-the-art systems, which have been heavily optimized by several approaches including multi-threading [62, 23, 51] and work-stealing scheme [51]. We attribute the performance issues mainly to the lim-

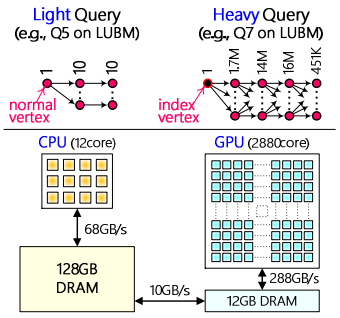

Fig. 3: Motivation of Wukong+G

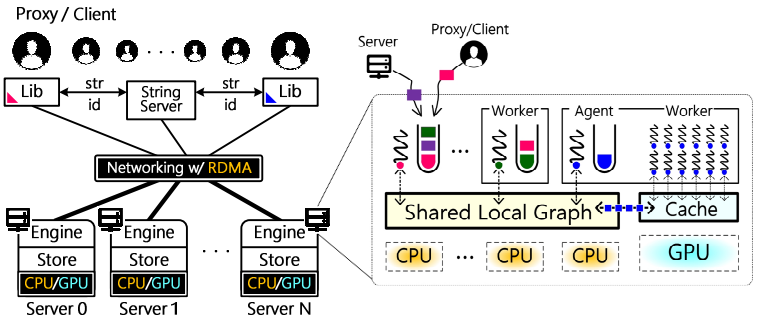

Fig. 4: The architecture overview of Wukong+G.

itation of handling hybrid workloads (light and heavy queries) on the homogeneous hardware (CPU), which can provide neither sufficient computation resources (a few cores) nor efficient data accesses (low bandwidth).

GPU is a good candidate to host heavy queries. First, the graph exploration strategy for query processing heavily relies on traversing massive paths on the graph store, which is a typical memory-intensive workload targeted by GPU's high memory bandwidth. Second, the memory latency hiding capability of GPU is inherently suitable for the random traversal on RDF graph, which is notoriously slow due to poor data locality. Third, every traversal path with the full-history pruning scheme is entirely independent, which can be fully parallelized on thousands of GPU cores.

In summary, the recent trend of hardware heterogeneity (CPU/GPU) opens an opportunity for running different queries on different hardware; namely, *running light queries on CPUs and heavy queries on GPUs*.

3 Wukong+G: An Overview

System architecture. An overview of Wukong+G's architecture is shown in Fig. 4. Wukong+G assumes running on a modern cluster connected with RDMA-capable fast networking, where each machine is equipped with one or more GPU cards. The GPU's device memory is treated as a cache for the large pool of the CPU's host memory. Wukong+G targets various SPARQL queries over a large volume of RDF data; it scales by partitioning the RDF graph into a large number of shards across multiple servers. Wukong+G may duplicate edges to make sure each server contains a self-contained subgraph (e.g., no dangling edges) of the input RDF graph for better locality. Note that there are no replicas of vertices in Wukong+G as no vertex data needs to synchronize. Moreover, Wukong+G also creates index vertices [51] for types and predicates to assist query processing.

Similar to prior work [51], Wukong+G follows a decentralized, shared-nothing, main-memory model on the server side. Each server consists of two separate layers: query engine and graph store. The query engine layer employs a worker-thread model by running N *worker threads* atop N CPU cores and dedicates one CPU core to run an *agent thread*; the agent thread will assist the worker threads on GPU cores to run queries. Each worker/agent thread on CPU has a task queue to continuously handle queries from clients or other servers, one at a time. The graph store layer adopts an RDMA-friendly key/value store over a distributed hash table to support a partitioned global address space. Each server stores a partition of the RDF graph, which is shared by all of worker/agent threads on the same server.

Wukong+G uses a set of dedicated proxies to run the client-side library and collect queries from massive clients. Each proxy parses queries into a set of stored procedures and generates optimal query plans using a cost-based approach. The proxy will further use the *cost* to classify a query into one of two types (light or heavy), and deliver it to a worker or agent thread accordingly.[4]

Basic query processing on GPU. In contrast to the query processing on CPU, which has to perform a triple pattern with massive paths in a verbose loop style (see Fig. 2), Wukong+G can fully parallelize the graph exploration with thousands of GPU cores. The basic approach is to dedicate one CPU core to perform the *control-flow* of the query, and use massive GPU cores to parallelize the *data-flow* of the query. As shown in Fig. 5, the agent thread on CPU core will first read the next triple pattern (❶) of the current query and prepare a cache of RDF datasets on GPU memory (❷). After that, the agent thread will leverage all GPU cores to perform the triple pattern in parallel (❸). Each worker thread on GPU core can independently fetch a row in the history table (❹) and combine it with the constant (takesCourse) of the triple pattern (TP-2) as the key (❺). The value retrieved by the key (❻) will be appended to a new column (?Z) of the history table (❼). While the hybrid design seems intuitive, Wukong+G still faces three challenges to run SPARQL queries on GPUs, which will be addressed by the techniques in §4:

[4]The recent release of Wukong (https://github.com/SJTU-IPADS/wukong) introduced a new cost-based query planner for graph exploration, where the cost estimation is roughly based on the number of paths may be explored. Wukong+G uses a user-defined threshold for the cost to classify queries.

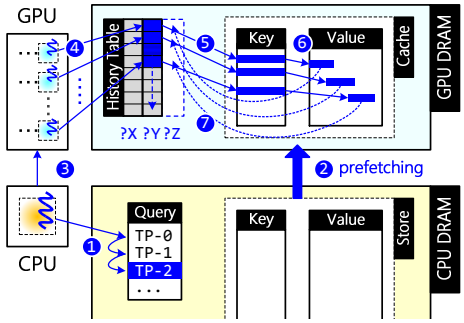

Fig. 5: The execution flow of query processing on CPU/GPU.

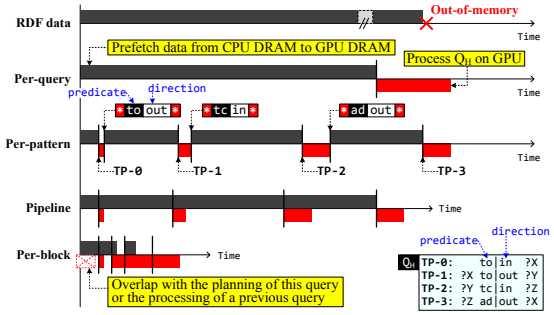

Fig. 6: The timeline of processing sample query (Q_H) on GPU.

C1: Small GPU memory. It is well known that GPU can obtain optimal performance only when the device memory (GPU DRAM) gets everything ready. Prior systems [62, 23, 51] can store an entire RDF graph in the host memory (CPU DRAM) since it is common that server-class machines equip with several hundred GBs of memory. However, this assumption does not apply to GPU since its current memory size usually stays less than 16GB. We should not allow device memory size to limit the upper bound of the supported working sets.

C2: Limited PCIe bandwidth. The memory footprint of SPARQL queries may touch arbitrary triples of the RDF graph. Therefore, the data transfer between CPU and GPU memory during query processing is unavoidable, especially for concurrent query processing. However, GPUs are connected to CPUs by PCIe (PCI Express), which has insufficient memory bandwidth (10GB/s). To avoid the bottleneck of data transfer, we should carefully design mechanisms to predict access patterns and minimize the number, volume and frequency of data swapping.

C3: Cross-GPU communication. With the increasing scale of RDF datasets and the growing number of concurrent queries, it is highly demanding that query processing systems can scale to multiple machines. Prior work [57, 51] has shown the effectiveness and efficiency of the partitioned RDF store and the worker-thread model. However, the intra-/inter-node communication between multiple GPUs has a long path: 1) device-to-host via PCIe; 2) host-to-host via networking; 3) host-to-device via PCIe. We should customize the communication flow for various participants to reduce the latency of network traffic.

4 Design

4.1 Efficient Query Processing on GPU

Facing the challenges like small GPU memory and limited PCIe bandwidth, we propose the following three key techniques to overcome them.

Query-aware prefetching. With the increase of RDF datasets, the limited GPU memory size (less than 16GB) is not enough to host the entire RDF graph. Wukong+G thus treats the GPU memory as a cache of CPU memory, and only ensures the necessary data is retained in GPU memory before running a query. However, it is non-trivial to decide the working set of a query accurately.

As shown in the second timeline of Fig. 6, Wukong+G proposes to just prefetch the triples with the *predicates* involved in a query, which can enormously reduce the memory footprint of a query from the entire RDF graph to the *per-query* scale. This assumption is based on two observations: 1) each query only touches a part of RDF graph; 2) the predicate of a triple pattern is commonly known (i.e., $\langle ?X, predicate, ?Y \rangle$). For example, the sample query (Q_H) only requires three predicates (teacherOf, takesCourse, and advisor), occupying about 3.7GB memory (0.3GB, 2.9GB, and 0.5GB respectively) for LUBM-2560.

Pattern-aware pipelining. For a query with many triple patterns, the total memory footprint of a single query may still exceed the GPU memory size. Fortunately, we further observe that the triple patterns of a query will be executed in sequence. It implies that Wukong+G can further reduce the demand for memory to the *per-pattern* scale. As shown in the third timeline of Fig. 6, Wukong+G can only prefetch the triples with a certain *predicate* that is used by the triple pattern will be immediately executed. Thus, for the sample query (Q_H) on LUBM-2560, the demand for GPU memory will further reduce to 2.9GB, the size of the maximum predicate (takesCourse).

Moreover, since the data prefetching and query processing are split into multiple independent phases, Wukong+G can use a *software pipeline* to create parallelism between the execution of the current triple pattern and the prefetching of the next predicate, as shown in the fourth timeline of Fig. 6. Note that it will also increase the memory footprint to the maximum size of two successive predicates (takesCourse and advisor).

Fine-grained swapping. Although the pattern-aware pipelining can overlap the latency of data prefetching and query processing, it is hard to perfectly hide the I/O cost due to limited bandwidth between system and

Table 1: A summary of optimizations for query processing on GPU. "$X|Y$" indicates XGB memory footprint and YGB data transfer. (†) The numbers are evaluated on 6GB GPU memory.

Granularity	Main Techniques	Q7 (GB) on LUBM-2560
Entire graph	Basic query processing	16.3 | 16.3
Per-query	Query-aware prefetching	5.6 | 5.6
Per-pattern	Pattern-aware pipelining	2.9 | 5.6
Per-block	Fine-grained swapping	2.9 | 0.7†

device memory (e.g., 10GB/s). For example, prefetching 2.9GB triples (takesCourse) requires about 300ms, which is even longer than the whole query latency (100ms). Therefore, Wukong+G adopts a fine-grained swapping scheme to maintain the triples cached in GPU memory. All triples with the same predicate will be further split into multiple fixed-size blocks, and the GPU memory will cache the triples in a best-effort way (§4.2). Consequently, the demand of memory will be further reduced to the *per-block* scale.

Moreover, the data transferring cost will also become the *per-block* scale, and all cached data on GPU memory can be reused by multiple triple patterns of the same query or even multiple queries. As shown in the fifth timeline of Fig. 6, when most triples of the required predicates have been retained in GPU memory, the prefetching cost can be perfectly hidden by query processing. Even for the first required predicate, Wukong+G still can hide the cost by overlapping it with the planning time of this query or the processing time of a previous query.

Table 1 summarizes the granularity of data prefetching on GPU memory, and shows the size of memory footprint and data transfer for a real case (Q7 on LUBM-2560). Note that Q7 is similar to Q_H but requires five predicates. The memory footprint of Q7 with fine-grained swapping is equal to the available GPU memory size (6GB) since Wukong+G only swaps out the triples of predicates on demand.

4.2 GPU-friendly RDF Store

Prior work [62, 51, 64] uses a distributed in-memory key/value store to physically store the RDF graph, which is efficient to support random traversals in graph-exploration scheme. In contrast to the intuitive design [62] that simply uses vertex ID (*vid*) as the key, and the in-/out-edge list (each element is a $[pid, vid]$ pair) as the value, Wukong [51] uses a combination of the vertex ID (*vid*), predicate ID (*pid*) and in/out direction (d) as the key (in the form of $[vid, pid, d]$), and the list of neighboring vertex IDs as the value (e.g., $[Logan, to, out] \longmapsto [DS]$ in the left part of Fig. 7).

This design can prominently reduce the graph traversal cost for both local and remote accesses. However, the triples (both key and value) with the same predicate are still sprinkled all over the store. It implies that the cost of prefetching keys and values for a triple pattern is extremely high or even impossible. Therefore, the *key/value store* on CPU memory should be carefully re-organized for heterogeneous CPU/GPU processing by aggregating all triples with the same predicate and direction into a *segment*. Furthermore, the key and value segments should be maintained in a fine-grained way (*block*) and be cached *in pairs*. Finally, the mapping between keys and values should be retained in the *key/value cache* on GPU memory, which uses a separate address space. Wukong+G proposes the following three new techniques to construct a GPU-friendly key/value store, as shown in the right part of Fig. 7.

Predicate-based grouping (CPU memory). Based on the idea of predicate-based decomposition in Wukong, Wukong+G adopts *predicate-based grouping* to exploit the predicate locality of triples and retains the encoding of keys and values. The basic idea is to partition the key space into multiple *segments*, which are identified by the combination of predicate and direction (i.e., $[pid, d]$). To preserve the support of fast graph exploration, Wukong+G still uses the hash function within the segment but changes the parameter from the entire key (i.e., $[vid, pid, d]$) to the vertex ID (*vid*). The number of keys and values in each segment are collected during loading the RDF graph and aligned to an integral multiple of the granularity of data swapping (*block*). To ensure that all values belonged to the triples with the same predicate are stored contiguously, Wukong+G groups such triples and inserts them together. Moreover, Wukong+G uses an indirect mapping to link keys and values, where the link is an *offset* within the value space instead of a direct pointer. As shown in the right part of Fig. 7, the triples required by TP-2 (i.e., $\langle Kurt, tc, DS\rangle$ and $\langle Bobby, tc, OS\rangle$) are aggregated together in both key and value spaces (the purple boxes).

Pairwise caching (GPU memory). To support fine-grained swapping, Wukong+G further splits each segment into multiple fixed-size blocks and stores them into *discontinuous* blocks of the cache on GPU memory, like $[Logan, to, out]$ and $[Erik, to, out]$. Note that the block size for keys and values can be different. Wukong+G follows the design on CPU memory to cache key and value blocks into separate regions on GPU memory, namely key cache and value cache. Wukong+G uses a simple table to map key and value blocks, and the link from key to value becomes the offset within the value block. Unlike the usual cache, the linked key and value blocks must be swapped in and out the (GPU) cache *in pairs*, like $[OS, tc, in]$ and $[Bobby]$ (the purple boxes). Thus, Wukong+G maintains a *bidirectional* mapping between the pair of cached key and value blocks. Moreover, a mapping table of block ID between RDF store (CPU) and cache (GPU) is used to re-translate the link between keys and values,

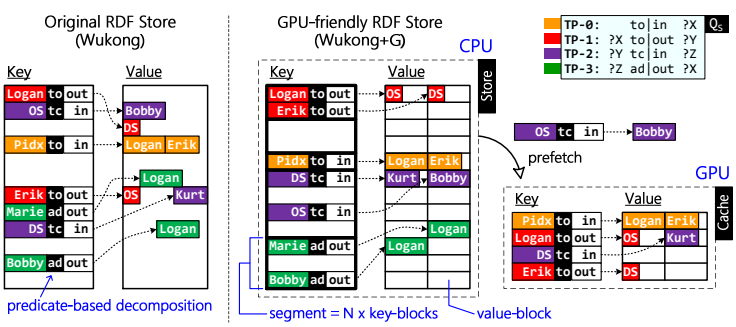

Fig. 7: The structure of GPU-friendly RDF store.

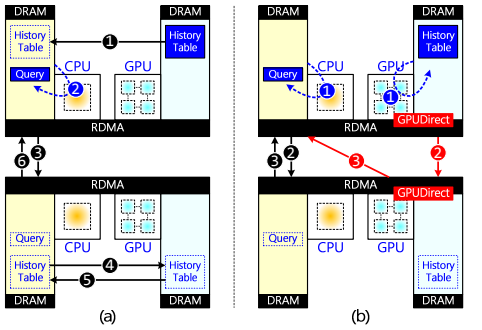

Fig. 8: Communication flow w/o and w/ GPUDirect.

when the pairwise blocks (key and value) are swapped in GPU memory.

Look-ahead replacement policy. The mapping table of block IDs between RDF store and cache records whether the block has been cached. Before running a triple pattern of the query, all of key and value blocks should be prefetched to the GPU memory. For example, the key $[OS, tc, in]$ and value $[Bobby]$ should be loaded into the cache before processing TP-2. Wukong+G proposes a *look-ahead LRU-based replacement policy* to decide where to store prefetched key and value blocks. Specifically, Wukong+G prefers to use free blocks first and then chooses the blocks that will not be used by the following triple patterns of this query (*look-ahead*), with the highest LRU score. The worst choice is the blocks will be used by the following triple patterns, and then the block of the farthest triple pattern will be replaced. Note that the replacement policies for keys and values are the same and there is at most a pair of key/value blocks will be swapped out due to the pairwise caching scheme.

For example, as shown in the right part of Fig. 7, before running the triple pattern TP-2, all key/value blocks of the predicate takeCourse (tc) should be swapped in the cache (the purple boxes). The value block with $[Bobby]$ can be loaded to a free block, while the key block with $[OS, tc, in]$ will replace the cached block with $[Pidx, to, in]$, since it was used by TP-0 with the highest LRU score.

4.3 Distributed Query Processing

Wukong+G splits the RDF graph into multiple disjoint partitions by a differentiated partitioning algorithm [51, 19] [5], and each machine hosts an RDF graph partition and launches many worker threads on CPUs and GPUs to handle concurrent light and heavy queries respectively. The CPU worker threads on different machines will only communicate with each other for (light) query processing, and it is the same to GPU worker threads for (heavy) query processing.

[5]The normal vertex (e.g., Logan) will be assigned to only one machine with all of its edges, while the index vertex (e.g., teacherOf) will be split and replicated to multiple machines with edges linked to normal vertices on the same machine.

To handle light queries on CPU worker threads, Wukong+G simply follows the procedure (see Fig. 2) that has been successfully demonstrated by Wukong [51]. However, to handle heavy queries on GPU worker threads, the procedure (see Fig. 5) becomes complicated due to the assistance of (CPU) agent thread and the maintenance of (GPU) RDF cache.

Execution mode: fork-join. Prior work [51] proposes two execution modes, *in-place* and *fork-join*, for distributed graph exploration to migrate data and execution respectively. The in-place execution mode synchronously leverages one-sided RDMA `READ` to directly fetch data from remote machines, while the fork-join mode asynchronously splits the following query computation into multiple sub-queries running on remote machines. Wukong+G follows the design on CPU worker threads but only adopts the fork-join mode for query processing on GPU, because the in-place mode is usually inefficient for heavy queries [51] and migrating data from remote CPU memory to local GPU memory is still very costly even with RDMA operations.

In the fork-join mode, the agent thread will split the running query (metadata) with intermediate results (history table) into multiple sub-queries for the following query processing, and dispatch them to the task queue of agent threads on remote machines by leveraging one-sided RDMA `WRITE`. Therefore, multiple heavy queries can be executed on multiple GPUs concurrently in a time-sharing way. However, the current history table is located in GPU memory (see Fig. 8), such that it would be inefficient to fetch and split the table by using a single agent thread on CPU (❶ and ❷ in Fig. 8(a)). Therefore, Wukong+G leverages all GPU cores to partition the history table in fully parallel (❶ in Fig. 8(b)) using a dynamic task scheduling mechanism [47, 18].

Communication flow. To support fork-join execution, the sub-queries will be sent to target machines with their metadata (e.g., query plan and current step) and history table (intermediate results), and the history table will be sent back with final results at the end. As shown in Fig. 8(a), the query metadata will be delivered by one-sided RDMA operations between the CPU memory of

Table 2: A collection of synthetic and real-life datasets. #T, #S, #O and #P mean the number of triples, subjects, objects and predicates respectively. (†) The size of datasets in raw NT format.

Dataset	#T	#S	#O	#P	Size†
LUBM-2560	352 M	55 M	41 M	17	58GB
LUBM-10240	1,410 M	222 M	165 M	17	230GB
DBPSB	15 M	0.3 M	5.2 M	14,128	2.8GB
YAGO2	190 M	10.5 M	54.0 M	99	13GB

two machines (❸ and ❻). In contrast, the history table has to go through a long path from local GPU memory to the remote GPU memory, and finally goes back to the local CPU memory. A detailed communication flow for history table (see Fig. 8(a)): 1) from local GPU memory to local CPU memory (❶, Device-to-Host); 2) from local CPU memory to remote CPU memory (❸, Host-to-Host); 3) from remote CPU memory to remote GPU memory (❹, Host-to-Device); 4) from remote GPU memory to remote CPU memory (❺, Device-to-Host); 5) from local CPU memory to remote CPU memory (❻, Host-to-Host).

GPUDirect [3] opens an opportunity for Wukong+G to directly write history table from local GPU memory to remote GPU and CPU memory. Hence, Wukong+G decouples the transferring process of query metadata and history table (❷ and ❷ in Fig. 8(b)), and further shortens the communication flow for history table by leveraging GPUDirect RDMA. It also avoids the contention on agent thread with the metadata transferring. A detailed communication flow for history table (see Fig. 8(b)): 1) from local GPU memory to remote GPU memory (❷, Device-to-Device); 2) from remote GPU memory to local CPU memory (❸, Device-to-Host).

Moreover, to mitigate the pressure on GPU memory when handling multiple heavy queries, Wukong+G choose to send the history table of pending queries from local GPU memory to the buffer on remote CPU memory first via GPUDirect RDMA, and delay the prefetching of history table from CPU memory to GPU memory till handling the query on GPU.

5 Implementation

Wukong+G prototype is implemented in 4,088 lines of C++/CUDA codes atop of the code base of Wukong. This section describes some implementation details.

Multi-GPUs support. Currently, it is not uncommon to equip every CPU socket with a separate GPU card for low communication cost and good locality. To support such multi-GPUs on a single machine, Wukong+G runs a separate server for each GPU card and several co-located CPU cores (usually a socket). All servers comply with the same communication mechanism via GPUDirect-capable RDMA operations, regardless of whether two servers share the same physical machine or not.

Too large intermediate results. In rare cases, the intermediate results may overflow the history buffer on GPU memory. For example, we witness this scenario in YAGO2 benchmark (§6.8) that a heavy query keeps spanning out without any pruning. Wukong+G can horizontally divide the intermediate results into multiple strips by row and only hold a single strip into the history table on GPU memory. The remaining strips will stay in CPU memory and be swapped in GPU memory one-by-one while processing a single triple pattern.

Table 3: The query performance (msec) on a single server.

LUBM-2560		TriAD	Wukong	Wukong+G
H	**Q1** (3.6GB)	851	992	**165**
	Q2 (2.4GB)	211	138	**31**
	Q3 (3.6GB)	424	340	**63**
	Q7 (5.6GB)	2,194	828	**100**
	Geo. M	639	443	**75**
L	**Q4**	1.45	**0.13**	0.16
	Q5	1.10	**0.09**	0.11
	Q6	16.67	**0.49**	0.51
	Geo. M	2.98	**0.18**	0.21

6 Evaluation

6.1 Experimental Setup

Hardware configuration. All evaluations are conducted on a rack-scale cluster with 10 servers on 5 machines. We run two servers on a single machine. Each server has one 12-core Intel Xeon E5-2650 v4 CPU with 128GB of DRAM, one NVIDIA Tesla K40m GPU with 12GB of DRAM, and one Mellanox ConnectX-3 56Gbps InfiniBand NIC via PCIe 3.0 x8 connected to a Mellanox IS5025 40Gbps IB Switch. Wukong+G only provides a one-to-one mapping between the work and agent threads on different servers [51], which mitigates the scalability issue of RDMA networks with reliable transports [31] and simplifies the implementation of the task queue. In all experiments, we reserve two cores on each CPU to generate requests for all servers to avoid the impact of networking between clients and servers as done in prior work [54, 56, 57, 20, 51].

Benchmarks. Our benchmarks include one synthetic and two real-life datasets, as shown in Table 2. The synthetic dataset is the Lehigh University Benchmark (LUBM) [7]. We generate 5 datasets with different sizes (up to LUBM-10240) and use the query set published in Atre et al. [13], which are widely used by many distributed RDF systems [36, 62, 23, 51]. The real-life datasets include the DBpedia's SPARQL Benchmark (DBPSB) [1] and YAGO2 [9, 30]. For DBPSB, we use the query set recommended by its official site. For YAGO2, we collect our query set from both H_2RDF+ [43] and RDF-3X [42] to make sure the test covers both light and heavy queries.

Comparing targets. We compare our system against two state-of-the-art distributed RDF query systems, TriAD [23] (RDF relational store) and Wukong [51] (RDF graph stores). Note that TriAD does not sup-

Table 4: The query performance (msec) on 10 servers.

LUBM-10240		TriAD	Wukong	Wukong+G
H	**Q1** (14.25GB)	3,400	480	**211**
	Q2 (9.74GB)	880	66	**12**
	Q3 (14.25GB)	2,835	171	**19**
	Q7 (22.58GB)	10,806	390	**100**
	Geo. M	3,094	215	**47**
L	**Q4**	3.08	**0.44**	0.46
	Q5	1.84	**0.13**	0.17
	Q6	65.20	**0.70**	0.71
	Geo. M	7.04	**0.34**	0.38

port concurrent query processing, so we only compare to it in the single query performance. As done in prior work [62, 23, 51], the string server is enabled for all systems to save memory usage, reduce network bandwidth, and boost string matching.

6.2 Single Query Performance

We first study the performance of Wukong+G for a single query using the LUBM dataset. Table 3 shows the optimal performance of different systems on a single server with LUBM-2560. For Wukong+G, there is no data swapping during single query experiment since the current memory footprint of all queries on LUBM-2560 (the numbers in brackets) is smaller than the GPU memory (12GB). The query-aware prefetching reduces the memory footprint to the per-query granularity (see Table 1).

Although Wukong and TriAD have enabled multi-threading (10 worker threads), Wukong+G can still significantly outperform such pure CPU systems for heavy queries (Q1-Q3, Q7) by up to 8.3X and 21.9X (from 4.5X and 5.2X) due to wisely leveraging hardware advantages. The improvement of average (geometric mean) latency reaches 5.9X and 8.5X. For the light queries (Q4-Q6), Wukong+G inherits the prominent performance of Wukong by leveraging graph exploration and outperforms TriAD by up to 32.7X.

We further compare Wukong+G with Wukong and TriAD (multi-threading enabled) on 10 servers using LUBM-10240 in Table 4. For heavy queries, Wukong+G still outperforms the average (geometric mean) latency of Wukong by 4.6X (ranging from 2.3X to 9.0X), thanks to the heterogeneous RDMA communication for preserving the good performance of GPU at scale. Further, using up all CPU worker threads to accelerate a single query is not practical for concurrent query processing since it will result in throughput collapse. For light queries, Wukong+G incurs about 12% performance overhead (geometric mean) compared to Wukong due to adjusting the layout of key/value store on CPU memory for predicate-based grouping. Wukong+G is still one order of magnitude faster than TriAD due to the in-place execution with one-sided RDMA `READ` [51].

6.3 Factor Analysis of Improvement

To study the impact of each technique and how they affect the query performance, we iteratively enable each optimization and collect the average latency by repeatedly running the same query on a single server with 3GB GPU memory for LUBM-2560. As shown in Table 5, even using query-aware prefetching (per-query), the memory footprints of query Q1, Q3 and Q7 still exceed available GPU memory (see Table 3). Hence, they can not run until enabling pattern-aware prefetching (per-pattern). The effectiveness of fine-grained swapping (per-block) varies on different queries. It is quite effective on Q2 and Q3 (8.8X and 5.0X) since all triples required by triple patterns can almost be stored in 3GB GPU memory. Note that Q3 returns an empty history table (intermediate results) half-way and reduces the practical runtime memory footprint to 2.5GB. For Q1 and Q7, although the relative large memory footprint (3.6GB and 5.6GB), incurs massive data swapping (1.5GB by 187 time and 5.1GB by 734 times), the cache sharing with fine-grained mechanism can still notably reduce the query latency by 2.4X and 1.4X. Moreover, pipeline does not work on Q2 and Q3 without data prefetching time. The improvement for Q1 and Q7 is still limited since the prefetching and execution time for each triple pattern are quite imbalanced. For example, 88% of blocks are swapped at two triple patterns for Q7.

Table 5: The contribution of (cumulative) optimizations to the query latency (msec) evaluated on 3GB GPU memory.

LUBM-2560	Per-query	Per-parttern	Per-block	Pipeline
Q1 (3.6GB)	x	743	313	295
Q2 (2.4GB)	284	283	32	31
Q3 (3.6GB)	x	309	62	63
Q7 (5.6GB)	x	893	622	610

Table 6: A comparison of query performance (msec) w/o and w/ GPUDirect RDMA (GDR) on 10 servers with LUBM-10240.

LUBM-10240	Q1	Q2	Q3	Q7
Wukong+G w/o GDR	222 (53.4)	13	22	103 (26.1)
Wukong+G w/ GDR	211 (40.1)	13	22	98 (21.3)

6.4 GPUDirect RDMA

To shorten communication flow and avoid redundant memory copy for history table (intermediate results) of queries, Wukong+G leverages GPUDirect RDMA (GDR) to write history table directly from local GPU memory to remote GPU and CPU memory (§4.3). To study the impact of leveraging GPUDirect RDMA, we enforce Wukong+G to purely use native RDMA for both query metadata and history table (i.e., Wukong+G w/o GDR). As shown in Table 6, the performance of Q2 and Q3 is non-sensitive to GPUDirect RDMA because of no data transfer among GPUs. For Q1, leveraging GPUDirect RDMA can reduce about 30% communication cost (53.4ms vs. 40.1ms), since the query need to send about 487MB intermediate results by about 990 times RDMA operations. For queries with relatively large intermediate results or many triple patterns, there are more rooms for the overall performance improvement.

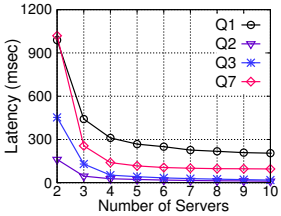

Fig. 9: The latency with the increase of servers.

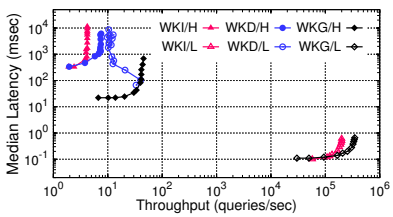

Fig. 10: The performance of hybrid workload on 10 servers with LUBM-10240.

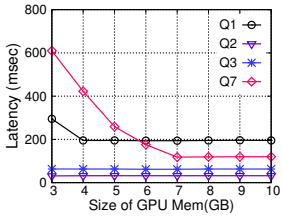

Fig. 11: The latency with the increase of GPU memory.

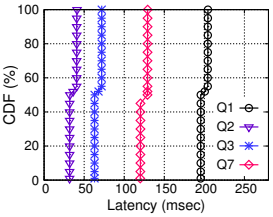

Fig. 12: The CDF of latency for mixed heavy workload.

6.5 Scalability

We evaluate the scalability of Wukong+G with the increase of servers. Since the latency of light queries of Wukong+G mainly inherits from Wukong, We only report the experimental results of heavy queries handled by GPUs. As shown in Fig. 9, the speedup of heavy queries ranges from 4.8X to 23.8X. As the number of servers increases from 2 to 10, a good horizontal scalability is shown. After a detailed analysis of the experimental results, we reveal that there are two different factors improving the performance at different stages. In the first stage (from 2 to 4 servers), the increase of total GPU memory provides the main contribution to the performance gains, ranging from 3.2X to 8.5X, by reducing memory swapping cost. In the second stage (from 4 to 10 servers), since Wukong+G stops launching expensive *memory swapping* operations when enough GPU memory is available, the main performance benefits come from using more GPUs, ranging from 1.5X to 2.8X.

Discussion. With the further increase of servers, the single query latency may not further decrease due to fewer workload per server and more communication cost. It implies that it is not worth making all resources (GPUs) participate in a single query processing, especially for a large-scale cluster (e.g., 100 servers). Therefore, Wukong+G will limit the participants of a single query and can still scale well in term of throughput by handling more concurrent queries simultaneously on different servers.

6.6 Performance of Hybrid Workloads

One principal aim of Wukong+G is to handle concurrent hybrid (light and heavy) queries in an efficient and scalable way. Prior work [51] briefly studied the performance of Wukong with a mixed workload, which consists of 6 classes of light queries (Q4-Q6 and A1-A3[6]). The light 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. Therefore, we first extend original mixed workload by adding 4 classes of heavy queries (Q1-Q3, Q7), and then allow all clients to freely send light and heavy queries[7] according to the distribution of query classes.

We compare Wukong+G (WKG) with two different settings of Wukong: Default (WKD) and Isolation (WKI). Wukong/Default (WKD) allows all worker threads to handle hybrid queries, while Wukong/Isolation (WKI) reserves half of the worker threads to handle heavy queries. Each server runs two emulated clients on dedicated cores to send requests. Wukong launches 10 worker threads, while Wukong+G launches 9 worker threads and an agent thread. The multi-threading for heavy queries is configured to 5. We run the hybrid workload over LUBM-10240 on 10 servers for 300 seconds (10s warmup) and report the throughput and median (50^{th} percentile) latency for light and heavy queries separately over that period in Fig. 10.

For heavy queries, Wukong+G improves throughput and latency by over one order of magnitude compared to Wukong (WKD and WKI). The throughput of WKD is notably better (about 80%) than that of WKI, since it can use all worker threads to handle heavy queries. For light queries, Wukong+G performs up to 345K queries per second with median latency of 0.6ms by 9 worker threads. The latency can be halved with a small 10% impact in throughput. As expected, WKI can provide about half of the throughput (199K queries/s) with a similar latency since only half of the worker threads (5) are used to handle light queries. However, the throughput and latency of WKD for light queries are thoroughly impacted by the processing of heavy queries.

6.7 RDF Cache on GPU

To study the influence of GPU cache for the performance of heavy queries on Wukong+G, we first evaluate the single query latency using LUBM-2560 on a single server with the GPU memory sizes varying from 3GB to 10GB. We repeatedly send one kind of heavy queries until the cache on GPU memory is warmed up, and illustrate the average latency of heavy queries in Fig. 11. Since the memory footprint of Q2 (2.4GB) is always smaller than the GPU memory, the latency is stable, and there is no data swapping. For Q3, although the memory footprint

[6]Three additional queries (A1, A2, and A3) are from the official LUBM website (#1, #3, and #5).

[7]In prior experiment [51], only up to one client is used to continually send heavy queries (i.e., Q1).

Table 7: The latency (msec) of queries on DBPSB and YAGO2

DBPSB	D1	D2	D3	D4	D5	Geo. M
Wukong	1.28	0.15	0.25	4.25	1.08	0.74
Wukong+G	0.53	0.16	0.26	0.99	0.52	0.41

YAGO2	Y1	Y2	Y3	Y4	Geo. M
Wukong	0.10	0.13	4685	752	14.6
Wukong+G	0.11	0.15	1856	398	10.5

of the query is about 3.6GB, the latency is still stable since the history table becomes empty after the first two triple patterns due to contradictory conditions, where the rest predicate segment (about 1.1GB) will never be loaded. For Q1 and Q7, the latency decreases with the increase of GPU memory due to the decrease of data swapping size. However, the break point of Q7 is later than that of Q1 since it has a relatively larger memory footprint (5.6GB vs. 3.6GB).

To show the effectiveness of sharing GPU cache by multiple heavy queries, We further evaluate the performance of a mixture of four heavy queries using LUBM-2560 on a single server with 10GB GPU memory. As shown in Fig. 12, the geometric mean of 50^{th} (median) and 99^{th} percentile latency is just 84.5 and 93.8 milliseconds respectively, under the peak throughput. Compared to the single query latency (see Table 3), the performance degradation is just 3% and 14%, thanks to our fine-grained swapping and look-ahead replacing. During the experiment, the number and volume of blocks swapped in per second are about 96 and 750MB.

6.8 Other Workloads

We further compare the performance of Wukong+G with Wukong on two real-life datasets, DBPSB [1] and YAGO2 [9]. As shown in Table 7, for light queries (D2, D3, Y1, and Y2), Wukong+G can provide a close performance to Wukong due to following the same execution mode and a similar in-memory store. For heavy queries (D1, D4, D5, Y3, and Y4), Wukong+G can notably outperform Wukong by up to 4.3X (from 1.9X).

7 Related Work

Wukong+G is inspired by and departs from prior RDF query processing systems [40, 58, 41, 12, 50, 13, 65, 60, 14, 61, 62, 23, 51, 32, 64], but differs from them in exploiting a distributed heterogeneous CPU/GPU cluster to accelerate heterogeneous RDF queries.

Several prior systems [11, 12] have leveraged column-oriented databases [53] and vertical partitioning for RDF dataset, which group all triples with the same predicate into a single two-column table. The predicate-based grouping in Wukong+G is driven by a similar observation. However, Wukong+G still randomly (hash-based) assign keys within the segment to preserve fast RDMA-based graph exploration, which plays a vital role for running light queries efficiently on CPU.

Using prefetching and pipelining mechanisms are not new, which have been exploited in many graph-parallel systems [49, 35] and GPU-accelerated systems [37] to hide the latency of memory accesses. Wukong+G employs a SPARQL-specific prefetching scheme and enables such techniques on multiple concurrent jobs (heavy queries) that share a single cache on the GPU memory.

There has been a lot of work [25, 24, 39, 26, 29, 27, 55, 46, 28] focusing on exploiting the unique features of GPUs to accelerate database operations. Mega-KV [63] is an in-memory key/value store that uses GPUs to accelerate index operations by only saving indexes on the GPU memory to ease device memory pressure. CoGaDB [15, 16] uses a column-oriented caching mechanism on GPU memory to accelerate OLAP workload. SABER [34] is a hybrid high-performance relational stream processing engine for CPUs and GPUs. Wukong+G is inspired by prior work, while the differences in workloads result in different design choices. To our knowledge, none of the above systems exploit distributed heterogeneous (CPU/GPU) environment, let alone using RDMA as well as GPUDirect features.

To reduce communication overhead between multiple GPUs, NVIDIA continuously puts forward GPUDirect technology [3], including GPUDirect RDMA and GPUDirect Async (under development [6]). They enable direct cross-device data transfer on data plane and control plane, respectively. Researchers have also investigated how to provide network [33, 21] and file system abstractions [52] based on such hardware features. Our design currently focuses on using GPUs to deal with heavy queries for RDF graphs. The above efforts provide opportunities to build a more flexible and efficient RDF query system through better abstractions.

8 Conclusions

The trend of hardware heterogeneity (CPU/GPU) opens new opportunities to rethink the design of query processing systems facing hybrid workloads. This paper describes Wukong+G, a graph-based distributed RDF query system that supports heterogeneous CPU/GPU processing for hybrid workloads with both light and heavy queries. We have shown that Wukong+G achieves low query latency and high overall throughput in the single query performance and hybrid workloads.

Acknowledgments

We sincerely thank our shepherd Howie Huang and the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61772335, 61572314, 61525204), the National Youth Top-notch Talent Support Program of China, and Singapore NRF (CREATE E2S2).

REFERENCES

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

[2] Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[3] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[4] Parallel Graph AnalytiX (PGX). http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/overview/index.html.

[5] Semantic Web. https://www.w3.org/standards/semanticweb/.

[6] State of GPUDirect Technologies. http://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf.

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

[8] Wikidata. https://www.wikidata.org.

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

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

[11] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB'07, pages 411–422, 2007.

[12] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, 2009.

[13] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 41–50, 2010.

[14] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 121–132, 2013.

[15] S. Breß. The design and implementation of CoGaDB: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[16] S. Breß, N. Siegmund, M. Heimel, M. Saecker, T. Lauer, L. Bellatreche, and G. Saake. Load-aware inter-co-processor parallelism in database query processing. *Data & Knowledge Engineering*, 93:60–79, 2014.

[17] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. Tao: Facebooks distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, pages 49–60, 2013.

[18] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT'10, pages 523–534, 2010.

[19] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1:1–1:15, 2015.

[20] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 26:1–26:17, New York, NY, USA, 2016. ACM.

[21] F. Daoud, A. Watad, and M. Silberstein. GPUrdma: Gpu-side library for high performance networking from gpu kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS'16, pages 6:1–6:8, 2016.

[22] Google Inc. Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[23] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. 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*, SIGMOD'14, pages 289–300, 2014.

[24] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[25] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, pages 511–524, 2008.

[26] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proceedings of the VLDB Endowment*, 6(10):889–900, Aug. 2013.

[27] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, Dec. 2014.

[28] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 1477–1492, 2015.

[29] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013.

[30] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference Companion on World Wide Web*, WWW'11, pages 229–232, 2011.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'16, 2016.

[32] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD'17, pages 1149–1164, 2017.

[33] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking abstractions for gpu programs. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, volume 14 of *OSDI'14*, pages 6–8, 2014.

[34] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD'16, pages 555–569, 2016.

[35] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *International Conference forHigh Performance Computing, Networking, Storage and Analysis*, SC'16, pages 830–841. IEEE, 2016.

[36] K. Lee and L. Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, Sept. 2013.

[37] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai. Garaph: efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 195–207. USENIX Association, 2017.

[38] National Center for Biotechnology Information. PubChemRDF. https://pubchem.ncbi.nlm.nih.gov/rdf/, 2014.

[39] S. I. F. G. N. Nes and S. M. S. M. M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *Data Engineering*, 40, 2012.

[40] T. Neumann and G. Weikum. RDF-3X: A risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, Aug. 2008.

[41] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 627–640, 2009.

[42] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[43] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2rdf+: High-performance distributed joins over large-scale rdf graphs. In *2013 IEEE International Conference on Big Data*, IEEE BigData'13, pages 255–263, 2013.

[44] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: Adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web*, WWW'12 Companion, pages 397–400, 2012.

[45] Philip Howard. Blazegraph GPU. https://www.blazegraph.com/whitepapers/Blazegraph-gpu_InDetail_BloorResearch.pdf, 2015.

[46] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*, ICDE'14, pages 508–519, 2014.

[47] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA'07, pages 13–24, 2007.

[48] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA'10, pages 4:1–4:5, 2010.

[49] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 472–488, 2013.

[50] S. Sakr and G. Al-Naymat. Relational processing of rdf queries: A survey. *SIGMOD Record*, 38(4):23–28, June 2010.

[51] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 317–332, 2016.

[52] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1):1:1–1:31, Feb. 2014.

[53] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[54] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 18–32. ACM, 2013.

[55] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.

[56] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[57] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[58] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, Aug. 2008.

[59] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 481–492, 2012.

[60] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD'12, pages 517–528, New York, NY, USA, 2012. ACM.

[61] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale rdf data. *Proceedings of the VLDB Endowment*, 6(7):517–528, May 2013.

[62] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276, 2013.

[63] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[64] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 614–630, 2017.

[65] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering sparql queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, May 2011.