Grasper: A High Performance Distributed System for OLAP on Property Graphs

Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, Xiao Yan
Department of Computer Science and Engineering
The Chinese University of Hong Kong
{hzchen,cjli,jcfang6,chhuang,jcheng,jzhang,yfhou,xyan}@cse.cuhk.edu.hk

ABSTRACT

The property graph (PG) model is one of the most general graph data model and has been widely adopted in many graph analytics and processing systems. However, existing systems suffer from poor performance in terms of both latency and throughput for processing online analytical workloads on PGs due to their design defects such as expensive interactions with external databases, low parallelism, and high network overheads. In this paper, we propose Grasper, a high performance distributed system for OLAP on property graphs. Grasper adopts RDMA-aware system designs to reduce the network communication cost. We propose a novel query execution model, called *Expert Model*, which supports adaptive parallelism control at the fine-grained query operation level and allows tailored optimizations for different categories of query operators, thus achieving high parallelism and good load balancing. Experimental results show that Grasper achieves low latency and high throughput on a broad range of online analytical workloads.

CCS CONCEPTS

• **Information systems → Online analytical processing engines**.

KEYWORDS

Distributed Systems, Graph Database, OLAP, RDMA

ACM Reference Format:
Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357223.3362715

1 INTRODUCTION

Graph analytics has a broad spectrum of applications in industry [49]. Recently, a large number of graph processing systems have been proposed [61, 66] (e.g., Pregel [40], PowerGraph [26]), which support efficient offline graph computation such as PageRank, BFS, and connected components. However, processing online graph analytical queries (or OLAP) remains to be challenging since it has much stricter requirements on both latency and throughput.

Among various graph data models, the **property graph** (**PG**) model is one of the most general and expressive representations because of the rich information it carries in addition to the pure graph topology. Each vertex/edge in a PG can have a label and a set of *properties* (or *attributes*) to describe the modeling entity/relationship. The properties allow various types of data to be stored and queried, which makes PG a very flexible model for connected data representation in practice. Because of this, more and more graph databases and OLAP graph systems have been developed for PGs, such as Neo4j [8], TinkerGraph [4], SQLGraph [55], JanusGraph [3], Titan [2], OrientDB [9] and TigerGraph [10]. In industry, companies such as Amazon and Alibaba also use PGs to store billion-level entities and relationships in various applications, while demanding online graph analytics with millisecond latency over a large PG [24].

Existing systems suffer from various performance problems due to the challenges of processing online analytical workloads on PGs. We analyze the limitations of existing systems in Section 3, and summarize the key challenging factors as follows. (1) *Diverse query complexity*: online analytical queries may differ significantly in their workloads (e.g., from a simple property check on a vertex to complicated pattern matching), and thus a mechanism is needed to support high parallelism and load balancing for heavy-workload queries, while at the same time preventing the starvation of light-workload queries. (2) *Diverse data access patterns*: query operators (e.g., *filter*, *traversal*, *count*) often have diverse access patterns on data and hence require different optimizations (e.g., cache, index), which makes it challenging to design a unified computational model that optimizes the execution of different query operators. (3) *High communication and CPU costs*: a query may have complex execution logics such as graph traversals that access a large portion of a PG, aggregation that collects intermediate results to one place through network, and joins that are CPU- and data-intensive. Such complex logics often result in high overheads on both communication and computation.

We propose **Grasper**, a distributed system designed to address the aforementioned challenges of processing online analytical queries on PGs. Grasper adopts *Remote Direct Memory Access* (*RDMA*) to reduce the high network communication cost for distributed query processing and introduces an RDMA-enabled *native PG storage* to exploit the benefits of RDMA. The key design of Grasper is a novel query execution model, called **Expert Model**, which achieves high utilization of CPU and network resources to maximize the processing efficiency. There are three core benefits brought by Expert Model: (1) it allows Grasper to support high concurrency in processing

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11...$15.00
https://doi.org/10.1145/3357223.3362715

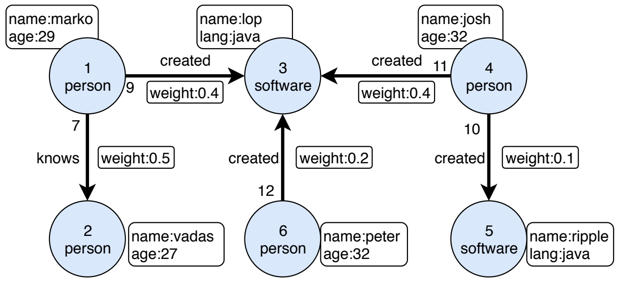

Figure 1: An example property graph

multiple queries and adaptive parallelism control within each query according to its workload; (2) it enables tailored optimizations for different categories of query operations based on their own data access patterns; (3) underlying system optimizations such as memory locality and thread-level load balancing are also incorporated into the design, which are critical for achieving millisecond latency for processing complex analytical queries.

We compared Grasper with popular systems such as Neo4j, JanusGraph, Titan, OrientDB and TigerGraph to highlight its unique designs. Grasper achieves competitive performance in terms of both query latency and throughput on benchmark workloads.

2 BACKGROUND

Property Graph. A property graph (PG) consists of a set of vertices V and a set of edges E, where vertices are entities and an edge models the relationship between two entities. We give an example in Figure 1. Each vertex in V has a label to represent its role (e.g., *person*, *software*) and a set of properties to describe its features in the form of *key-value* pairs, e.g., (*name*, "*marko*"), (*age*, *29*). Different vertices may have different properties without a fixed schema. Edges are usually directed and each edge also keeps a label and a set of properties. In many applications, vertices have characteristic properties such as name, age, gender, etc., while edges have quantitative properties such as weight, cost, distance, etc.

Query Languages. Several graph query languages have been proposed, such as Gremlin [7], Cypher [5] and SPARQL [45]. Cypher is a declarative query language mainly used in Neo4j [8]. SPARQL is the standard query language for RDF data, which has an SQL-like recursive interface matched with the RDF protocol. Gremlin is a procedural language supported by Apache TinkerPop [4], which allows users to succinctly express queries as a set of **query steps** (also called **pipes**). Among them, Gremlin is currently the most popular query language adopted by existing graph databases and query systems [4, 43] as it is easy to use Gremlin to write a graph query using intuitive expressions such as high-level traversals, exact pattern defining (e.g., *match()*), and even programmatical primitives (e.g., *repeat()*, *until()*). We adopt Gremlin in Grasper.

RDMA. Remote Direct Memory Access (RDMA) can bypass the OS and supports zero-copy transmission. There are two types of commands for remote memory access, two-sided *send/recv* verbs and one-sided *read/write* verbs. In one-sided verbs, the memory address in a remote machine where a message is to be sent is set by the sender side, and the RDMA operations complete without any knowledge of the remote process. As one-sided RDMA *read/write* verbs can provide higher bandwidth and lower latency than the two-sided operations, they are more popular in high-performance distributed system designs such as [15, 53, 72].

The performance characteristics of one-sided RDMA *read* and *write* have been well studied [41, 52]. Compared with Ethernet and IPoIB (IP over InfiniBand), RDMA can achieve at least an order of magnitude lower round-trip latencies (< 10us) when the payload is small (i.e., < 4K bytes). In addition, the latency of RDMA is relatively insensitive to the increase in the payload size for small-sized messages, as long as the network is not saturated. To be more specific, in our RDMA-enabled cluster, the latency of one-sided read operation increases only 2.2X (from 3.94us to 8.63us) when the payload size increases 512X (from 8 bytes to 4K bytes). When the payload continues to grow (> 4K bytes), the latency of RDMA starts to increase linearly, which is similar to Ethernet and IPoIB, but RDMA is still an order of magnitude faster than Ethernet and >2X faster than IPoIB. RDMA also achieves much higher throughput under all payload sizes, which is about 10X higher than Ethernet and IPoIB for both small (< 4K bytes) and large payloads (> 4K bytes).

3 MOTIVATION

To find the limitations of existing systems and motivate the design of Grasper, we report the query latency and throughput of two popular distributed graph databases, Titan v.1.1.0 [2] and JanusGraph v.0.3.0 [3]. We used LDBC [23], which is a recognized benchmark for graph data management [28, 31]. We used a synthetic large graph (i.e., LDBC in Table 3) created by the LDBC-SNB Data Generator, and ran the experiments on a 10-node cluster (see configuration in Section 6).

We first show that it is challenging for Titan and JanusGraph to achieve low latency for processing complex analytical workloads. We used one typical complex query IC4 (i.e., Interactive Complex 4), which has relatively heavy workload. Figure 2 reports the breakdown of the query latency of IC4 for each query step. The time spent on the query steps varies significantly. In particular, *hasLabel* (i.e., a filter on entities based on their label) and *in* (i.e., a filter-based traversal on the graph topology) took up most of the query processing time. Due to the diverse execution logics and data access patterns of different query steps, it is common to have large variation in the latency of processing different query steps of complex analytical queries. Thus, the underlying query execution system should be better to *adaptively change the parallelism for executing different query steps according to their workloads*, which we found lacking in existing systems, hence resulting in either high latency for processing expensive query steps (thus high latency for the whole query) or wasting resource for processing simple query steps (due to fixed parallelism).

Next, we examine if Titan and JanusGraph are able to efficiently use resources to achieve high throughput. We used a mixed workload consisting of IS1-IS4 (i.e., Interactive Short) queries and fed them to the systems continuously for a period of 180 seconds. While we report the detailed comparison on the throughput in Section 6, here we discuss the CPU and network utilization of the systems. Under a saturated throughput, Figure 3 shows that both systems recorded low utilization of CPU and network. This is mainly caused by the non-native graph storage and inefficient query execution model (especially for processing multiple queries concurrently), where similar

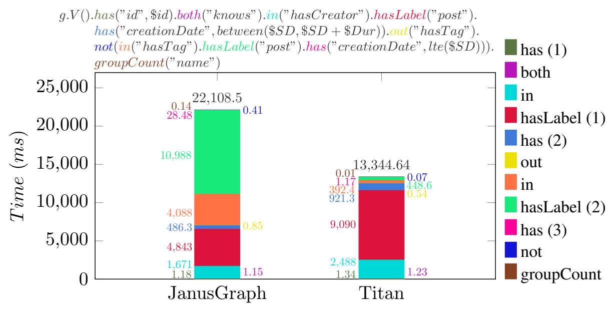

Figure 2: [Best viewed in color] The query latency breakdown of IC4 in the LDBC benchmark

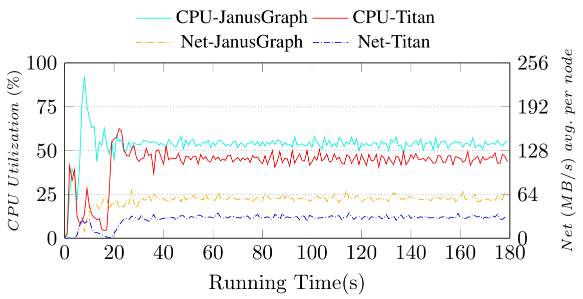

Figure 3: [Best viewed in color] CPU and network utilization for a mixed workload {IS1-IS4}

conclusions have also been made in [43]. Both Titan and JanusGraph follow the one-query-one-thread mechanism, and they partition the PG data among servers and use a third-party relational database (e.g., HBase) as the underlying storage. Such an architecture leads to extra overheads on the interaction between the storage layer and the execution layer. In addition, they adopt a non-native graph model (i.e., using the relational representation model for graphs), which is not efficient for traversal-based operations that are common in online analytical graph queries, e.g., searching neighborhoods starting from some vertices, path-based queries, expanding a clique, etc.

Design Goals. Based on the above analysis, we come up with the following design goals. (1) The system should explore the characteristics of OLAP workloads on PGs in its design, in particular, an efficient query execution model to achieve high utilization on both CPU and network. This motivates us to design Expert Model to support efficient processing of various analytical operations (e.g., traversals, filters, and aggregations). (2) Due to the diversity of OLAP queries, Expert Model should achieve high CPU utilization with both *high parallelism* (within a machine and across machines) for processing a heavy-workload query (to achieve low latency) and *high concurrency* for processing as many queries simultaneously as possible (to achieve high throughput). (3) We should avoid using an external database or data store, but integrate the data store with the query execution engine tightly to eliminate unnecessary overheads, and data storage should be native for graph representation. (4) Since network communication is critical for distributed systems, we can make use of RDMA to reduce the cost of network communication. Accordingly, the designs of data store and system components should be tightly associated with RDMA-related features.

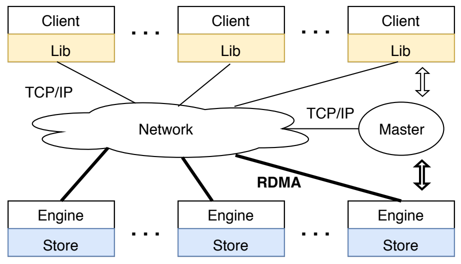

Figure 4: The architecture of Grasper

4 SYSTEM DESIGN

Overview. We first show the architecture of Grasper in Figure 4. A Grasper cluster consists of a set of query servers and a set of clients. Query servers are connected by RDMA network. Each query server can receive and process multiple queries concurrently with a pool of worker threads. Grasper supports an arbitrary number of clients, which are connected to query servers through TCP/IP network. A client involves only a light-weight communication lib used to submit query strings and receive query results.

A query server consists of two key components: **data store** and **query engine**. The distributed data stores of all query servers maintain the partitioned PG data across multiple machines. To handle the diverse access patterns of different query operators, we adopt a heterogeneous storage to keep the graph topology and the vertex/edge properties separately (Section 4.1). The query engines are responsible for query parsing and query execution (Section 4.3). In particular, we propose a novel Expert Model to support concurrent query execution with optimizations tailored for each category of query operations. To reduce query latency and maximize CPU utilization for various query workloads, Expert Model enables adaptive parallelism control on query processing both within a query and across queries.

Grasper also assigns a master node to monitor the progress of query execution on each query engine through a heartbeat mechanism and accordingly to coordinate the assignment of incoming queries from clients to servers based on the current load of each engine.

4.1 Data Store

A PG consists of not only the graph topology but also vertex/edge labels and properties. We may use relational, wide-column or other NoSQL databases to store graph data. However, these data stores are not natively designed for storing graph data, which may break data locality and incur costly multi-way joins on graph traversal based operations. To enable a *query-friendly* data layout, we propose a hybrid, native graph storage, which jointly considers the data access patterns of graph OLAP workloads and the performance characteristic of RDMA.

We divide the in-memory space of each Grasper server into two parts: *Normal Memory* and *RDMA Memory*. Normal Memory is used to store those graph data that are accessed locally during query execution, i.e., graph topology and intermediate results generated by queries. RDMA Memory is pre-registered with a fixed size and used to store the data that could be remotely accessed by other query engines. Figure 5 gives an illustration using the example PG in

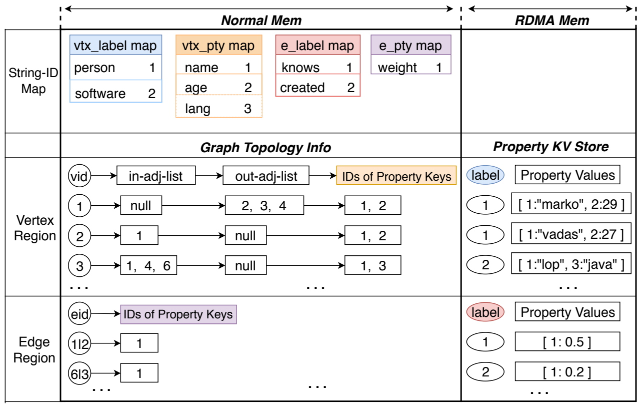

Figure 5: Data store in Grasper

Figure 1. The *String-ID Map* in Normal Memory maps all strings in the labels and property keys into unique IDs after data loading, in order to save network bandwidth and memory consumption. *Vertex Region* stores (in Normal Memory) the graph topology information of the local partition of vertices, including the vertex ID, the directed adjacency lists (i.e., in-neighbors and out-neighbors), and the IDs of the property keys (for property value retrieval). RDMA Memory in all query servers are connected together to form a *distributed RDMA-enabled key-value store* (*KVS*) for storing vertex labels and property values, which can be remotely requested by queries. *Edge Region* has almost the same layout as Vertex Region but stores the data of the local edges in Normal Memory and the edge labels and property values in RDMA Memory.

In such a hybrid data storage design, we use index-free adjacency lists to support efficient traversal-based query operations and use RDMA-enabled KVS to achieve low-cost remote access to labels and property values through one-sided RDMA *read*. We keep the graph topology data in Normal Memory since Grasper never directly accesses remote partitions of the graph topology, as each query engine only executes query steps on its own local partition and cooperates with remote engines by wrapping "*non-local graph traversals*" into *messages* sent through one-sided RDMA *write* (Section 4.2).

Note that since partitioning properties alongside with their vertices/edges would lead to uneven storage and accordingly cause load imbalance among query engines, we partition vertices/edges (in Normal Memory) and their properties (KVS in RDMA Memory) separately to form independent storage spaces. We give more details about graph partitioning in Section 5.

We show the overall memory layout of a Grasper server in Figure 6. In Normal Memory, in addition to the graph topology store, we also allocate a zone called *Index Buff* to record the index maps of properties whose indexes have been built (see index construction in Section 5). Another zone called *Meta Heap* is set to maintain those temporary data necessary for query processing such as metadata of queries and intermediate results. In RDMA Memory, besides the two KV-stores (i.e., V-KVS, E-KVS) for V/E properties and the *Meta Heap* zone which has the same function as the one in Normal Memory but for RDMA, we also allocate two zones, *Send Buffs* and *Recv Buffs*, for message management in RDMA communication (Section 4.2).

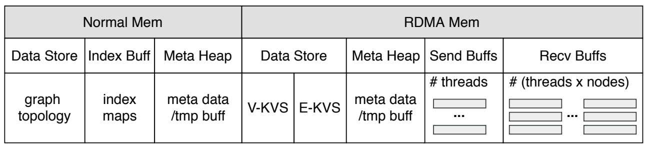

Figure 6: Memory layout on a Grasper server

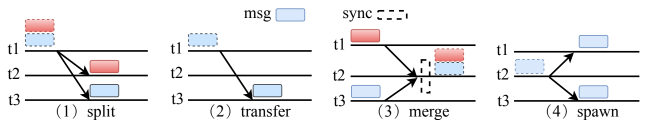

Figure 7: RDMA message dispatching in Grasper

4.2 RDMA Communication

Grasper uses RDMA-based, thread-level communication channels for message passing among the worker threads in servers. We only use one-sided RDMA *read/write* because of their superior performance [20, 42, 54]. In each query server, there are w *SendBufs* and a $(w \times n)$ 2D *RecvBufs* to form an *RDMA Mailbox*, where w is the number of worker threads in a query server and n is the total number of query servers. Each *SendBuf*$[i]$ loads the sending messages of a worker thread for RDMA communication, and each *RecvBufs*$[i][j]$ keeps those messages that are received from query server j and to be consumed by worker thread i for efficient query processing without locking. We implement each RecvBufs$[i][j]$ as a ring buffer adopted from FaRM [21].

In Grasper, we define a **message** as one piece of intermediate or final query result generated from one specific query step together with the metadata (e.g., step type). A message is the basic communication unit among all worker threads, and we impose an upper bound on its size to benefit from the short RDMA latency with small payloads (Section 2). Consequently, when the intermediate result is too large, we will split it into multiple messages. This procedure is called *message splitting*. In addition, Grasper also has three other message dispatching patterns for thread-level communication: *transfer*, *merge*, and *spawn*, as presented in Figure 7. These four RDMA message dispatching patterns are used in various *query steps* for query processing.

4.3 Query Engine

To address the limitations of existing graph systems for processing online analytical queries, we carefully design the query engine following design goals (1) and (2) presented in Section 3.

We also remark that, although the distributed data stores and RDMA-based communication channels in Grasper achieve design goals (3) and (4), the novelty and key contributions of our work lie in the design of the query engine and Expert Model.

4.3.1 Query Plan Construction. Grasper adopts Gremlin as the query language but provides its own query execution plan. Instead of only assigning one worker thread to one query at a time as in existing graph systems [2, 3, 9], Grasper adopts adaptive parallelism control based on the workload of a query step. We define a concept, **flow type**, to describe the flow pattern of each query step according to its functionality. There are three flow types in Grasper: **sequential**, **barrier**, and **branch**, which use the message dispatching patterns,

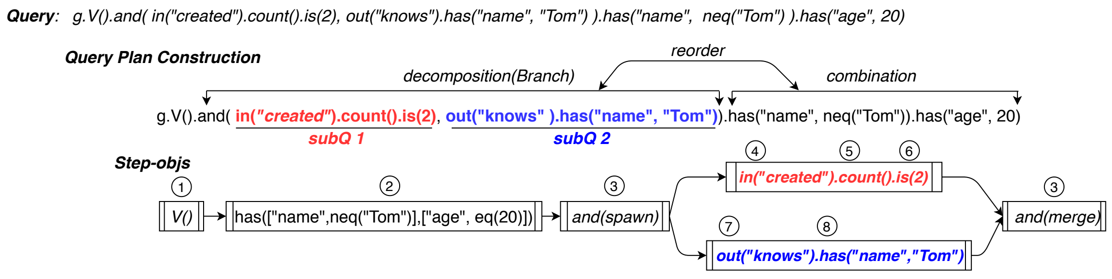
Query: g.V().and(in("created").count().is(2), out("knows").has("name", "Tom")).has("name", neq("Tom")).has("age", 20)
Query Plan Construction
decomposition(Branch)
reorder
combination
g.V().and(in("created").count().is(2), out("knows").has("name", "Tom")).has("name", neq("Tom")).has("age", 20)
subQ 1
subQ 2
Step-objs
V()
has(["name",neq("Tom")],["age", eq(20)])
and(spawn)
in("created").count().is(2)
out("knows").has("name","Tom")
and(merge)

Figure 8: Query plan construction

transfer, *merge*, and *spawn+merge*, respectively. We define each of the flow types as follows:

- **Sequential**: A *sequential* query step receives a message, processes it, and generates new message(s) to be processed by the next step. The traversal-based and filter-based query steps (e.g., *in*, *out*, *has*) are usually sequential. Multiple messages may be received by a sequential step and as messages of a sequential step are independent of each other, they can be processed in parallel by multiple threads.
- **Barrier**: A query step of this type collects messages from the previous step and performs a functional aggregation on the collected data before proceeding to the next step. *Barrier* steps always need synchronization before moving forward (e.g., *count*, *max*).
- **Branch**: A *branch* query step (e.g., *or*, *and*, *union*) creates multiple sub-query branches, where the message received by the step is duplicated as the input of the sub-queries for parallel processing. The results of these sub-query steps are finally merged back into one query step.

When a query is received from a client, the master assigns it to a query engine, which becomes the **coordinator** of the query. The coordinator parses the query string into an ordered list of *step_objs*, where each *step_obj* is an instance of a query step and maintains the meta data of the step such as the step type, the associated parameters, and the position of the next *step_obj* to be processed. We follow a tree-based parsing rule to recursively parse the query string step by step in order to construct the logical query plan according to the query optimization rules. Although the optimization of query execution plan is out of scope of our paper, and there have been extensive studies [11, 16, 25, 46, 51], we briefly introduce the following three optimization rules that we apply in Grasper for query plan construction. We illustrate the idea using an example query in Figure 8.

- **Decomposition**: to decompose *branch*-type query steps (e.g., *and*, *or*, *union*) into multiple sub-queries, which can process in parallel. In Figure 8, the *and()* step is decomposed into $subQ_1$ and $subQ_2$.
- **Combination**: to combine contiguous filter-based steps on vertices/edges/properties into one single step, which reduces the redundant overhead of multiple scanning and filtering. In Figure 8, the two *has()* steps are combined into one *step_obj*.
- **Reordering**: inspired by the optimizations on join [14, 35, 44], to reorder query steps with specific constraints and move them to the front so as to generate less intermediate results, thus reducing the costs of both computation and communication. In the example query, we move the combined *has()* step to the front, as shown in Figure 8.

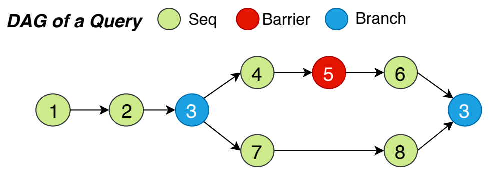

Figure 9: The DAG of the query in Figure 8

The constructed query plan of the example query is shown at the bottom of Figure 8. Note that all the *Branch* type should be split into two parts for message *spawn* (e.g., step 3 is split to start the execution of the sub-queries, i.e., steps 4-6 and steps 7-8, in parallel) and message *merge* (to collect the intermediate results to one query engine for the processing of the next step). Similarly, the *Barrier* type also needs to collect the shuffled intermediate results from all query engines to the coordinator engine for data aggregation.

Each query step belongs to only one flow type. Thus, based on the message dependency and execution order in the query plan, we can generate a directed acyclic graph (i.e., DAG) for each query to describe its parallelism. We give an example of DAG in Figure 9 for the query in Figure 8. Then, Expert Model will enable more specific optimizations for a query based on its DAG structure.

4.3.2 Expert Model. Expert Model is the central idea of Grasper, which defines the distributed execution workflow for each query and is the key to Grasper's high performance.

Design Philosophy. Expert Model provides a top-down query-specific mechanism and considers the characteristics of OLAP workloads to address the limitations presented in Section 3 and achieve low latency and high throughput. Expert Model supports the following three features to improve query performance: (1) adaptive parallelism control at step-level inside each query; (2) tailored optimizations for various query steps according to their characteristics; (3) locality-aware thread binding and load balancing. In particular, we integrate these features into each **expert**, which is a physical query operator in Grasper, to allow fine-grained specialization for various query optimizations.

Intuitively, an **expert** can be considered as a unique executor that is responsible for the processing of one specific category of query steps that share similar functionalities (e.g., *has*, *hasKey* and *hasValue*), where each expert has its own unique optimizations associated with a category of query steps. Note that an expert may

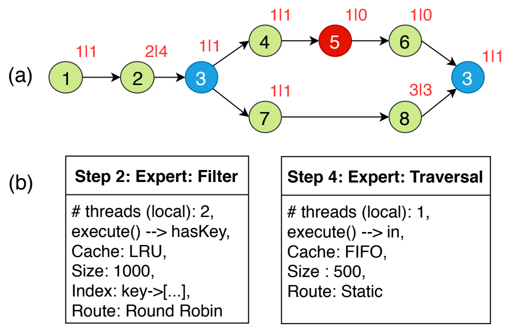

Figure 10: (a) adaptive parallelism at step-level; (b) an expert example

employ multiple threads to concurrently process the query steps of multiple queries with shared optimizations. Feature (1) in Expert Model ensures that the runtime parallelism degree of an expert is adaptive according to the workload. Feature (2) in Expert Model improves the efficiency of query processing, and Feature (3) handles the underlying cache locality and load balancing among the threads physically. In this way, each feature takes care of one aspect for query performance, while these features are specialized in each expert for each specific category of query steps.

Expert Model is designed to be a general model. Although in Grasper we select Gremlin as the query language, the design of Expert Model can be implemented and extended to other query processing systems with their own query languages. In the following discussion, we first introduce the three features mentioned above, and then present the detailed mechanism of experts.

Adaptive Parallelism Control. Inside a query, those query steps with heavy workloads (e.g., *in*, *has*) usually become the performance bottleneck, as we showed in Section 3. To address this problem, Grasper has a mechanism to adjust the execution parallelism of each query step adaptively according to its workload. The workload of a query step is measured as the number of messages it receives at each query engine. As the size of each message is bounded (Section 4.2), an expert can simply set the parallelism (i.e., the number of worker threads) as the number of messages. As an illustration, Figure 10(a) shows the parallelism of each query step of the DAG in Figure 9. In this example, we consider only two query engines and each node in the DAG has its own runtime parallelism on the two engines, e.g., "2|4" on step 2 means that engine 1 and engine 2 assign 2 and 4 threads to process the *hasKey* operator in parallel. This also indicates that the load of *hasKey* on engine 2 is heavier than that on engine 1. Note that the parallelism of the *barrier*-type step 5 is 1 globally because of its aggregate function. In constrast, both *sequential* and *branch* query steps have no limitation on parallelism.

Tailored Optimizations for Query Steps. There are groups of query steps that have similar functionalities. For example, filter-based query steps such as *has*(), *hasKey*() and *hasValue*() have almost the same data access pattern. In addition, we also consider locality in our design, as similar query steps tend to access the same shard of graph data in a query server (i.e., spatial locality) and consecutive queries may share some common query steps (i.e., temporal locality). Therefore, we group those query steps that share similar functionalities and access patterns into one category of executor,

Table 1: The expert pool in Grasper

Expert	Query Steps
Init	g.V(), g.E()
End	N/A [to aggregate the final results]
Traversal	in, out, both, inE, outE, bothE, inV, outV, bothV
Filter	has, hasNot, hasKey, hasValue
Range	range, limit, skip
Order	order
...	...
Group	group, groupCount
Math	min, max, mean,
BranchFilter	and, or, not

i.e., an expert, and accordingly to apply tailored optimizations (e.g., specific caching strategy, cache size, message routing rules, etc.) on different experts to achieve high query efficiency. The overall resource utilization of the system will also be improved through such optimizations tailored for each category of query steps, as compared with applying optimizations to individual query steps separately and maintaining their own private states, which may lead to poor caching [29] and memory fragmentation [12].

Locality-Aware Thread Binding and Load Balancing. Expert Model employs a thread pool in each query engine for the parallel execution of one query and the concurrent processing of multiple queries. As each expert has its own data structures (e.g., cache) that occupy memory space, thread switching on OS and NUMA architecture will lead to extra overheads and negative side-effects [33, 47] on query latency. We propose a locality-aware thread binding strategy to address this concern. Specifically, we bind each thread to one CPU core physically and then divide threads in the thread pool into several regions logically according to the topology of CPUs in each NUMA node and the number of NUMA nodes per machine. Each expert is assigned to one region only to ensure that there is no thread switching and across-node memory access. The load balancing among worker threads is another important problem, as the overloading on one thread may create a straggler in the processing of a query. Hence, Expert Model allows each expert to manage its own message routing (i.e., to send each message to which thread for next processing) with a global view of the thread loading, so that the messages of a query step can be distributed evenly among the threads in the corresponding region.

The Mechanism of Experts. An **expert** is a physical query operator that *expertly* handles the processing of all query steps belonging to one category (say, *C*). Each expert maintains its own data structures (e.g., indexes, cache) for tailored optimizations, its own *execute()* function, and its own routing rules for out-going messages, to handle the *concurrent* processing of query steps in *C*. In each Grasper server, we allow only one expert instance launched for each category *C*. As a consequence, all query steps belonging to one category will be processed by its unique expert only, with shared data structures for tailored optimizations. For example, when remote access is needed (e.g., for processing query step *Values*), or CPU-intensive calculation is conducted (e.g., for processing query step *Group*), the corresponding expert can use cache to avoid redundant communication or computation, where the cached data is shared for all worker threads.

Table 2: Thread regions with core binding

Thread Region	Experts
Cache_Seq	Filter, FilterLabel, Key, Values, Properties, Label
Cache_Barrier	Group, Order
Traversal	Traversal, Init
Normal_Seq	Is, As, Select, Where
Normal_Barrier	Count, Range, Math, Aggregate, Cap, Dedup
Normal_Branch	Branch, BranchFilter

The kernel of Grasper's query engine is composed by a set of experts, which forms an *expert pool*. As an illustration, Table 1 lists some typical experts [1] as well as the query steps associated with each of them. In addition, based on the in-memory access patterns (e.g., flow type, whether using cache or not), we divide the experts into six thread regions, as listed in Table 2, in order to achieve the locality-aware thread binding.

As a result, we create a top-down mechanism to provide Grasper the ability of fine-grained query optimization control according to the characteristic and the processing load of a query at runtime. Using cache as an example, we choose to apply LRU cache in a read-heavy expert, *Filter*, to avoid repeated remote access on property values, since the LRU strategy is more likely to retain those items that are frequently and recently accessed. Instead, we prefer to apply FIFO cache in the expert *Traversal* for caching vertex/edge labels, considering that label access normally happens in query steps such as $g.v().In().Out()$, for which the FIFO strategy is more suitable in such a BFS-like access pattern.

Figure 10(b) provides more details based on an example query in Figure 8. In Step 2, the *Filter* expert is processing the *hasKey* query step with a parallelism of 2 on query engine 1. It maintains an LRU cache with 1,000 slots and an index for fast retrieval of property keys. Once the execution is completed, the *Filter* expert will send all newly generated messages to the query engines for the processing of next step, where the query engines store the data needed for the next step. Here, the *Filter* expert applies the round-robin strategy to route the messages to idle threads in the *thread region* which the next expert belongs to, for efficient memory access in the same NUMA node. In addition, when we continue to process this query from Step 2 (i.e., *hasKey*) to Step 8 (i.e., *has*), the *Filter* expert will be launched again but with a different parallelism adapted to the workload of Step 8. In Step 4, the *Traversal* expert is activated to traverse all in-neighbors (i.e., the *in* query step) of the input vertices with a parallelism of 1 on query engine 1. At this time, it holds an FIFO cache with size 500 and routes the new messages using a static strategy (i.e., to the coordinator directly), since its next step (i.e., *count*) is a Barrier type.

Work Flow. When a query engine is launched, its expert pool will be initialized and all experts will be constructed and kept alive until the engine shuts down. Figure 11 depicts the architecture of Expert Model, where the expert pool is built upon the core-bound thread pool, and each thread has its own *RecvBufs* to receive messages. When a thread becomes idle, it will try to read one message from its RecvBufs. If it succeeds, the *expert dispatcher* will parse this

[1]Due to the page limit, we cannot list them all in here. More details can be found on the wiki page of Grasper's Github repository.

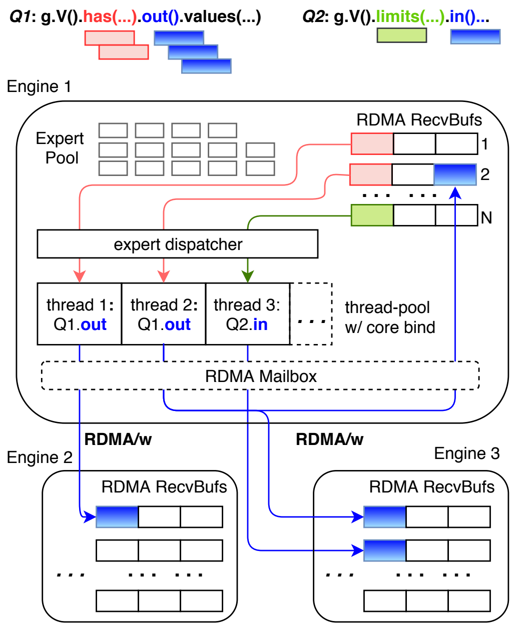

Figure 11: The architecture of Expert Model

message to obtain its step type and invoke the *execute()* function of the corresponding expert to process this query step.

Expert Model can process multiple queries concurrently. We illustrate using Q1 and Q2 in Figure 11. The *has*() step of Q1 generates 2 messages (colored in pink) and the *limits*() step of Q2 generates 1 message (colored in green). Then, all these 3 messages are processed in parallel by 3 threads, i.e., threads 1-2 (working on Q1.*out*) and thread 3 (working on Q2.*in*). These 3 threads are employed by the *Traversal* expert with shared cache and indexes as well as other specific optimizations. That is, the cached data, indexed data (if any), and other data structures of the *Traversal* expert are being utilized by these threads through internal sharing. Accordingly, the CPU utilization and memory locality can be both improved. To distribute the newly generated messages (colored in blue) evenly among threads in the receiving query engines, the *Traversal* expert applies its own routing rule to determine the specific location (i.e., thread and engines) of each message.

5 IMPLEMENTATION & OPTIMIZATIONS

Grasper was implemented in C++, currently with 18K+ lines of code (https://github.com/yaobaiwei/Grasper). The clients connect to the master and servers using ZeroMQ TCP sockets. The servers use MPI to coordinate their inter-process communication for graph data loading and shuffle. The RDMA initialization and one-sided *read/write* between servers are based on the *librdma* library.

Graph Partitioning. Grasper loads the source PG data from HDFS, where a query server reads a part of the data. Then, we partition and shuffle the entities of the graph topology (i.e., V and E) into the data store and the entities of the properties on V and E into the KVS, separately according to their IDs. We choose a hash partitioning strategy to achieve evenly data loading. Although partitioning the graph topology separately with the vertex/edge properties may seem to result in more network communication in query processing, this is actually a better strategy for the following reasons. First, in our design, as the network is not a bottleneck thanks to RDMA, we care more about the balanced distribution of data which directly impacts on the CPU utilization. Second, if we partition the graph topology and properties together (i.e., vertices and edges, along with their properties, are distributed to the same query servers), then the distribution of the properties would be rather uneven among query servers because the number of properties varies considerably among different vertices/edges, which could lead to more serious performance problems (e.g., stragglers) due to unbalanced loads.

Index Construction. Indexes are important for the efficient processing of query steps such as *has*, *hasLabel* and *hasValue*. Grasper supports various types of common indexes on text and numerical values for fast look-up or range search on vertex/edge labels and property values. Users can specify the type of index to build and the indexable keys (i.e., which property to index) through an interactive client console. Then, a special expert, *Index*, will be invoked on each query engine to construct the corresponding index simultaneously. The *Index* expert also creates an index map for the various indexed keys, and then hands over the indexes as well as the index maps to the respective experts for processing its associated query steps (e.g., an index on the labels of V will be handed over to the *Filter* expert for processing the query step *hasLabel*).

RDMA-Enabled KVS. The RDMA-enabled KVS was implemented based on DrTM [59], which splits the storage space into two regions: *header* and *entry*. The header consists of N buckets, each of which contains multiple header slots. A slot is the basic unit in the header, which maintains a pointer that records the corresponding offset and length of the value in the entry region. The KVS has a fixed size in RDMA Memory, where the user-defined capacity should be set large enough to hold the property values for the whole graph.

Work Stealing. The diversity of OLAP workloads on PGs may lead to stragglers. In Grasper, the stragglers are mainly overloaded worker threads as network communication is not the bottleneck anymore due to the use of RDMA (as our experiments in Section 6 show). Grasper provides a mechanism for idle threads to steal messages from the *RecvBufs* of other busy threads to improve CPU utilization and reduce query latency. Based on the thread regions and the topology of NUMA nodes, we define a stealing priority order: an idle thread first checks the *RecvBufs* of the other threads in the same thread region, and then those of the other regions in the same NUMA node, and finally those of the other NUMA nodes, until it obtains a message to process.

6 EXPERIMENTAL EVALUATION

The experiments were run on a cluster of 10 machines, each with two 8-core Intel Xeon E5-2620v4 2.1GHz processors and 128GB of memory. Each machine is also equipped with a Mellanox ConnectX-3 40Gbps Infiniband HCA. The machines run on CentOS 6.9 with the OFED 3.2 Infiniband driver. For fair comparison, we used 24 computing threads in each machine for all systems we compared with, and tried our best to tune their configuration (i.e., system parameters) to the setting that gives their best performance. All results reported in the paper are the average of five runs.

Table 3: Dataset statistics

Dataset	$\|V\|$	$\|E\|$	$\|VP\|$	$\|EP\|$
LDBC	59,308,744	357,617,104	321,281,654	101,529,501
AMiner	68,575,021	285,667,220	291,161,548	120,381,452
Twitter	52,579,682	1,963,262,821	320,732,961	577,955,736

Table 4: Query benchmark

Q1	g.V().has([*filter*]).properties([*property*])
Q2	g.V().hasKey([*filter1*]).hasLabel([*label*]).has([*filter2*])
Q3	g.V().has([*filter*]).in([*label*]).values([*key*]).max()
Q4	g.E().has([*filter1*]).outV().dedup().has([*filter2*]).count()

Q5	g.E().has([*filter1*]).not(outV([*label*]).has([*filter2*]))
	.groupCount([*key*])

Q6	g.V().has([*filter*]).and(
	    out([*label1*]).values([*key1*]).min().is([*predicate1*]),
	    in([*label2*]).count().is([*predicate2*])
	).values([*key2*])

Q7	g.V().has([*filter1*]).as('a').union(
	    out([*label1*]),
	    out([*label2*]).out([*label3*])
	).in([*label4*]).where(neq('a')).has([*filter2*])
	.order([*property*]).limit([*number*])

Q8	g.V().has([*filter1*]).aggregate('a').in([*label1*]).out([*label2*]). .has([*filter2*]).where(without('a'))

Benchmarks. We used the LDBC social network benchmark [23], which is the currently best known benchmark for graph query workloads. We generated a synthetic property graph using the LDBC-SNB data generator. For the query workload, we select 8 typical ones from two scenarios: Interactive Complex IC1 - IC4, and Interactive Short IS1 - IS4.

We also evaluated the systems on a real-world property graph, AMiner[2] from Open Academic Society[3]. Since the LDBC benchmark queries can only be evaluated on its own synthetic dataset, we followed the design principles of LDBC to carefully create a benchmark for OLAP on a general PG. The benchmark was designed based on the following five assessment criteria:

(1) **Query complexity**: starting from simply *finding a certain property on some vertices* and gradually raising the complexity to *finding a subgraph involving multi-hop traversals with a certain pattern*.
(2) **Scope of data access**: some queries start from only one vertex or one subgraph, while others start from the whole graph.
(3) **Query result size**: the size of the intermediate/final results of some queries is fixed (e.g., aggregation queries), while that of others varies significantly depending on factors such as the neighborhood size and the graph size.
(4) **Diversity of query steps**: the query steps cover a wide range of functional types and flow types.
(5) **Execution cost**: the queries cover heavy, medium and light workloads.

We list our benchmark in Table 4, from which we created the query workloads by sampling some property values and labels in the dataset for the arguments (e.g., [*filter*], [*property*], [*label*], [*key*]) in the eight query templates [4]. We also give the statistics of LDBC and AMiner in Table 3, listing the number of vertices, edges, vertex properties and edge properties, respectively.

[2]https://academicgraphv1wu.blob.core.windows.net/aminer/aminer_papers_0.zip
[3]https://www.openacademic.ai/
[4]The specific keys and values used in the templates will be released on our project homepage.

Table 5: Query latency (in *msec*) of distributed systems

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	271	16.7	388	77.3	0.30	2.19	0.91	0.32
Titan	66985	13585	5.8E5	11947	0.71	25.9	2.88	1.32
J.G.	56206	9223	4.5E5	22420	0.83	14.5	2.99	1.17
AMiner	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
Grasper	0.17	0.42	17.3	45.2	104	28.8	2.32	4.41
Titan	1.07	12.4	32341	2.1E5	43809	234	9.11	84.08
J.G.	1.34	8.70	27466	2.4E5	39155	276	5.61	84.71

Table 6: Query latency (in *msec*) of single-machine systems

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	1935	75.1	2550	223	0.48	2.51	1.38	0.13
Neo4J	1448	372	15042	293	20.7	77.6	16.3	21.7
OrientDB	32869	2140	20721	2582	0.91	25.1	3.46	1.47
T.G.(install + run)	46517 +55.3	40739 +18.2	44048 +117	43685 +30.1	37745 +8.03	41629 +11.1	38799 +9.39	37708 +7.66

6.1 Query Latency

We first analyze the query latency, comparing with Titan v.1.1.0 [2], JanusGraph v.0.3.0 [3], Neo4j v.3.5.1 [8], OrientDB v.3.0.6 [9] and TigerGraph Developer Edition [10]. These systems are the currently best known systems for processing online analytical queries on PGs. In this experiment, each system only processed a single query each time so as to make sure that the query latency was not affected by the processing of other queries.

We first report the results of comparing with the distributed systems, Titan and JanusGraph, running on 10 machines and using the same number of threads as Grasper. The latency of processing each type of query is shown in Table 5. Grasper achieves excellent performance compared with Titan and JanusGraph for all types of queries on both benchmarks, especially for processing the heavy-workload queries (e.g., IC1-IC4, Q3, Q4, Q5). These queries have heavy workloads because they have complex query logics (e.g., IC1-IC4) or the size of the intermediate results between query steps is large (e.g., Q3-Q5), which are challenging for existing systems to achieve low query latency. For example, Titan and JanusGraph do not have optimizations specifically designed for processing heavy-workload query steps and there is also non-trivial communication overheads with the external systems (i.e., HBase and Elasticsearch) on which they are built. In contrast, Grasper integrates the data store with the query engine, and further reduces the communication cost through RDMA. As RDMA shifts the bottleneck of distributed query processing from network to CPU computation, Expert Model supports adaptive parallelism to better utilize CPU threads and provides better in-memory locality through tailored optimizations, both of which are essential to achieve low latency for processing heavy query steps.

To further demonstrate the effect of an integrated design, Figure 12 reports the breakdown of the query latency of Grasper for processing IC4, as well as the CPU and network utilization of a mixed workload consisting of IS1-4 queries for a period of 180 seconds. Compared with the results of Titan and JanusGraph reported in Figures 2 and 3 in Section 3, Grasper only needs at most 60ms to process the bottleneck steps (i.e, *hasLabel*, *in*), while Titan and JanusGraph took more than 10 seconds to process them. In addition, the CPU and network utilization have also been significantly improved to around 95% and 380+ MB/s, respectively, due to our integrated design and Expert Model.

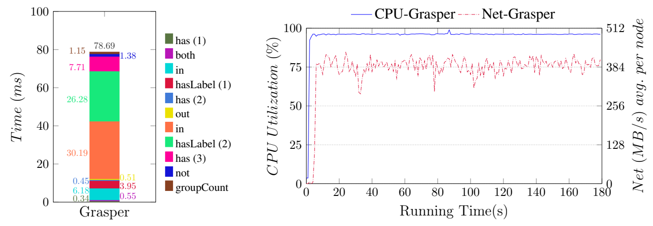

Figure 12: [Best viewed in color] (a) The query latency breakdown of IC4 on LDBC by Grasper; (b) CPU and network utilization of Grasper for the mixed workload {IS1-IS4}

Next we report the results of comparing Grasper with Neo4j, OrientDB and TigerGraph, where all the four systems were deployed on a single machine and using the same number of threads. We only ran the experiment on the LDBC benchmark since for the AMiner dataset, Neo4j ran into errors during index construction and OrientDB ran out of memory during data loading. Table 6 lists the query latency of the systems. Grasper achieves very competitive performance especially compared with Neo4j and OrientDB. Compared with TigerGraph, Grasper has advantage in processing IS1-IS4. For IC1-IC4, TigerGraph's run time is very competitive, which is significantly shorter than the query time of all the other systems. This is because TigerGraph processes a query first by an "install" process, which (according to their documentation) pre-translates and optimizes the query, and then by a "run" process to execute the installed query. The install process takes a long time, but significantly improves the run time for all types of queries. As TigerGraph is not fully open source, we cannot further analyze what exactly TigerGraph does in installing a query. Note that TigerGraph can also be run in an "interpret" mode, which directly processes a query without installing the query, but its latency is significantly longer than Grasper and Neo4J for processing any of the queries. Overall, the results also show that Neo4j performs better on complex queries (i.e., IC1-IC4), OrientDB is better on simple queries (i.e, IS1-IS4), TigerGraph has competitive run time but at the cost of a long install process, and Grasper achieves good balanced performance for all queries without an expensive installation or optimization process.

As there is no network overhead in the single-machine setting, this experiment on a single machine also shows that Grasper's good performance does not come only from its use of RDMA, but also comes from its other unique designs such as Expert Model, which we will further validate in Section 6.3. The results of Table 5 and Table 6 also show that 10 machines (more parallelism) can significantly speed up the processing of the complex queries that have heavy workloads, thus justifying the need for distributed query processing (even though the graph can fit into a single machine).

6.2 Query Throughput

As an OLAP system, the throughput performance is also important. In one of the world's largest e-commerce companies and a financial company we collaborate with, tens of thousands to a hundred thousand queries (the majority are simple ones) can be submitted every second at peak time. Currently, they achieve the required throughput using a large number of fast machines but this approach is having trouble catching up with their business growth rates. Grasper is designed to achieve high throughput for handling such a situation.

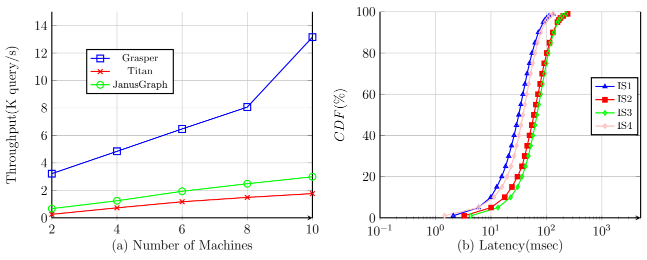

Figure 13: [Best viewed in color] (a) Throughput on LDBC for {IS1-IS4}; (b) CDFs of Grasper's query latency for {IS1-IS4} (using 10 machines)

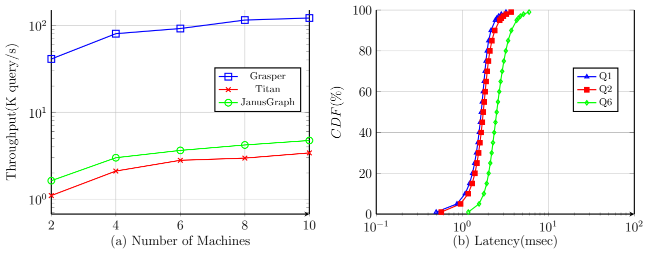

Figure 14: [Best viewed in color] (a) Throughput on AMiner for {Q1, Q2, Q6}; (b) CDFs of Grasper's query latency for {Q1, Q2, Q6} (using 10 machines)

To evaluate query throughput performance, we used the IS queries in the LDBC benchmark and {Q1, Q2, Q6} in our benchmark as the query templates to generate as many queries as the systems could handle. Figure 13 and Figure 14 report the throughput comparison with the other two distributed systems. As Figure 13(a) shows, Grasper achieves a throughput of 13K+ queries/sec on 10 machines for the LDBC workload. As a comparison, JanusGraph's throughput is 2.9K+ queries/sec and Titan's is 1.7K queries/sec. Compared with JanusGraph and Titan, Grasper also achieves good scalability as its throughput increases more than 4 times when the number of machines increases from 2 to 10. Figure 14(a) shows that on our benchmark using the AMiner dataset, Grasper's throughput is about 20-30 times higher than that of JanusGraph and Titan. The gap between Grasper's throughput and that of JanusGraph and Titan is much bigger for this workload than for the LDBC workload because the queries generated from the {Q1, Q2, Q6} templates have lighter workload than the LDBC queries, as also reflected by Grasper's throughput on these two types of workloads. Grasper achieves much higher throughput for processing queries of lighter workload because the design of its Expert Model enables high concurrent processing of multiple queries: first, adaptive parallelism control sets a minimum parallelism needed for processing a light workload so that threads can be more fully used; second, tailored optimizations effectively share common data structures for the processing of many query steps that belong to the same category, thus allowing resources to be better utilized to process more queries.

Figure 13(b) and Figure 14(b) further plot the CDF curves of each class of queries processed by Grasper. The results show that the 50th percentile latency is only 2-5 times shorter than the 99th percentile latency. These steep curves and the absence of long tail in the CDFs also indicate that there is no starvation phenomenon

Table 7: Query latency (in *msec*) of Grasper w/ and w/o APC

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	271	16.7	388	77.3	0.30	2.19	0.91	0.32
w/o APC	469	24.8	666	131	0.51	3.63	1.43	0.54
AMiner	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
Grasper	0.17	0.42	17.3	45.2	104	28.8	2.32	4.41
w/o APC	0.20	0.62	23.7	59.6	111	35.4	4.50	6.15

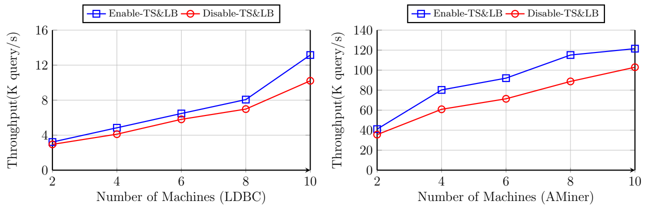

Figure 15: (a) Throughput of Grasper on LDBC for {IS1-IS4}, and (b) throughput of Grasper on AMiner for {Q1, Q2, Q6}, w/ and w/o TS&LB

when we process a large number of queries concurrently. The results verify the effectiveness of Expert Model in handling load balancing (or stragglers) and high concurrent processing.

6.3 Effects of System Designs & Opts

In this set of experiments, we want to evaluate the effects of Grasper's system designs and optimizations on its performance. Note that it is difficult to measure the performance benefits brought by each component/technique in Grasper as often we only see a positive effect when different components/techniques are integrated together. Thus, we only tested those parts that are easier to be evaluated on their individual effects.

Table 7 first reports the effect of *adaptive parallelism control* (*APC*) on query latency by comparing the two cases when Grasper enables/disables APC, where disabling APC means to process message (if any) with fixed parallelism of 1 in each server. The results show that APC achieves speed-ups in processing all the queries, especially for the complex queries IC1-IC4 that have the heaviest workloads, because their bottleneck steps are given higher parallelism for their execution.

Then we examined the effect of locality-aware thread binding and load balancing (TS&LB) on query throughput, using the same workloads as in Section 6.2. Figure 15 shows that the overall throughput of Grasper is considerably improved after the TS&LB mechanism is enabled. This indicates that the memory locality on CPU, the side-effects of thread switching and stragglers due to overloaded threads affect the throughput performance. Grasper's Expert Model provides an integrated design to address these issues.

We further tested the effects of other important designs of Grasper, including RDMA, query plan optimization (i.e., Q.Opts) and work stealing (i.e., Steal). We disabled them one by one, where -RDMA means the use of IPoIB instead of RDMA on InfiniBand, -Q.Opts and -Steal mean without using Q.Opts or Steal, respectively. As reported in Table 8, RDMA can indeed reduce the latency of all queries, because even the simplest query needs to frequently access the properties on remote servers through the network. We remark that the performance improvement also comes from Grasper's RDMA-friendly designs that enable higher CPU utilization by taking advantage of

Table 8: Query latency (in *msec*) of [Grasper-X] (using 10 machines)

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	271	16.7	388	77.3	0.30	2.19	0.91	0.32
-RDMA	1349	17.97	1253	260	1.04	2.57	2.06	1.26
-Q.Opts	374	19.39	558	81.26	0.31	2.38	0.93	0.32
-Steal	488	24.68	671	127	0.57	3.25	1.31	0.54
AMiner	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
Grasper	0.17	0.42	17.3	45.2	104	28.8	2.32	4.41
-RDMA	0.54	1.18	21.54	70.47	222	30.69	9.09	6.23
-Q.Opts	0.17	4254	22.84	417	131	35.49	2.89	4.34
-Steal	0.23	0.61	20.91	57.62	111	33.44	4.01	6.07

one-sided RDMA *read/write*, as we showed in Figure 12(b). While RDMA is effective, other optimizations also contribute to Grasper's overall high performance. The work stealing technique brings the greatest contribution to the latency improvement of processing IC2 and IS2. Especially for IC2, the benefit brought by RDMA is more limited due to the heavy computation workload of IC2. Query plan optimization is critical (far more important than RDMA or Steal) for processing queries generated from the Q2 and Q4 templates, both of which have multiple filter-based operators. For processing these queries, the latency is reduced as much as from 4,254ms to 0.42ms, and from 417ms to 45.2ms, respectively, through query step combination and reordering.

In conclusion, each individual part evaluated in this subsection has their own effects on Grasper's performance. Although the benefits brought by some parts may seem to be (relatively) smaller, it is by combining the benefits of all parts together that makes Grasper a high performance system.

6.4 Scalability

Finally, we evaluated the scalability of Grasper for processing queries with different workloads. For the eight queries in the LDBC benchmark, they are classified into two groups: IC1-IC4 have *heavy* workloads and IS1-IS4 have *light* workloads. For the eight queries in our own benchmark, we also classify them into two groups: {Q1, Q2, Q8} have *light* workloads due to highly selective filters, and {Q3, Q4, Q5, Q6, Q7} have *heavy* workloads with both relatively high CPU computation (i.e., *max*, *order*, *where*) and network communication (i.e., large intermediate results).

We increased the number of machines from 2 to 10 to measure the change on query latency. Figures 16(a) and 16(c) report the results for the IS queries on LDBC and the light queries on our benchmark using the AMiner dataset. Processing these queries has relatively stable performance and using more machines does not reduce the latency as it is sufficient to process these light-workload queries with a small amount of computing resources. Comparatively, Figure 16(b) and 16(d) show that the latency of processing the heavy-workload queries can decrease quite significantly (note that the figures are in log scale) when more machines are used. This is because the complex logic and heavy computation can benefit from more CPU resource and higher parallelism, where more machines/servers also mean more experts to process the query steps in parallel. Note that the traversal-based and filter-based query steps in these queries, which belong to the *sequential* flow type, can achieve more speed-ups with higher parallelism. Although the cost of communication will increase when we use more machines, the network communication cost is no longer the major overhead due to the use of RDMA.

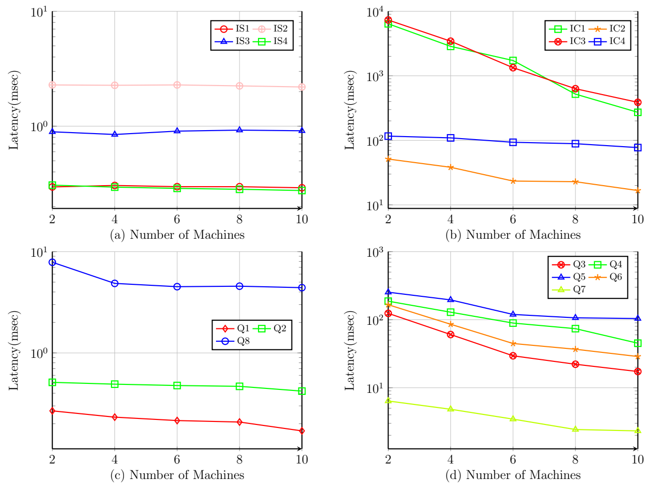

Figure 16: The scalability of Grasper on query latency

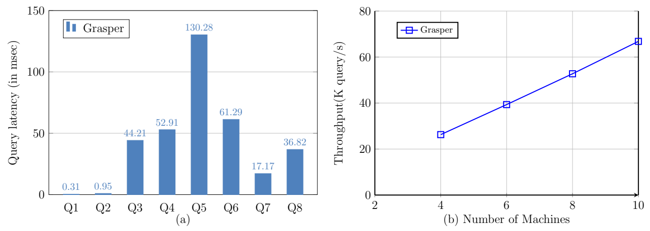

Figure 17: (a) Query latency (in *msec*) of Q1-Q8 on Twitter by Grasper (using 10 machines); (b) Throughput of Grasper on Twitter for {Q1, Q2, Q6}

We further evaluate the system performance on a much larger graph. We used the Twitter [5] graph, which originally has no property attached, and we randomly generated some properties for both vertices and edges. Some statistics of Twitter is also given in Table 3. For this graph, only Grasper can be run and both Titan and JanusGraph could not finish data loading even in 3 days using 10 machines. Figure 17 reports the results of Grasper for query latency and throughput. Note that Grasper crashed on 2 machines due to OOM issue. On this much larger dataset, Grasper also achieves high performance with short query latency (in milliseconds) and linear throughput. The results show the competitiveness of Grasper on processing large graphs compared with existing graph databases.

6.5 Comparison with RDF Systems

The Resource Description Framework (RDF) is another popular format to express the entities and relationships in real-world. It usually models the knowledge graph as a list of triplets (i.e., subject, predicate, object). Logically, the PG and RDF can be converted into each other. For this reason, we also compared with two RDF querying systems, Wukong [52] and Virtuoso [22]. Wukong is the state-of-the-art RDF querying system and significant performance improvements over existing systems are reported in [52]. It also uses RDMA and thus serves as a good comparison for Grasper. Virtuoso is

[5]http://konect.uni-koblenz.de/networks/twitter_mpi

Table 9: Query latency (in *msec*) of Grasper vs. RDF systems on DBPedia (using 10 machines)

DBpedia	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Grasper	0.19	0.20	22.24	97.78	939	7.32	29.23	417
Wukong	0.22	-	157	-	-	-	-	-
Virtuoso	0.38	0.86	122	1527	4494	42.20	65.80	2028

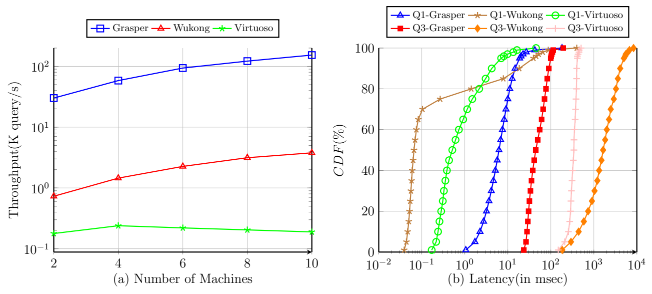

Figure 18: [Best viewed in color] (a) Throughput on DBPedia for {Q1, Q3}; (b) CDFs of query latency on DBPedia (using 10 machines)

a well-known RDF query systems built on a hybrid RDBMS/Graph column store.

We used a RDF graph DBpedia[6] (including two parts: *Citation Data* and *Citation Links*) for this experiment. When the dataset is converted from RDF to PG, there is a 25% decrease on the data size, which indicates that PG model is more compact than RDF. As Wukong and Virtuoso use SPARQL as the query language, we converted the Gremlin queries into equivalent SPARQL queries. However, currently Wukong could only process Q1 and Q3, but not the other queries, due to the lack of associated SPARQL operators (e.g., *Exist*, *Union*).

Table 9 shows that Grasper achieves shorter latency on all queries, and the performance gap significantly widens for those queries with heavier workloads (e.g., Q4, Q5, Q8). This validates the importance of specialized system designs and optimizations for the efficient processing of complex queries, regardless of the underlying storage model. We further used {Q1, Q3} as templates to generate a mixed query workload for throughput evaluation. Figure 18(a) shows that Grasper also achieves high throughput compared with Wukong and Virtuoso. The higher throughput may be explained by Grasper's better resource utilization as the CDF curves in Figure 18(b) show that for Wukong, more than 25% of the queries with light workloads (i.e., Q1) have much longer latency than the rest, and Virtuoso's case is also not much better. In contrast, the difference in query latency is much smaller for Grasper, as Expert Model processes each step with adaptive parallelism and tailored optimization, while also enabling idle worker threads to steal work from others for better CPU utilization.

7 RELATED WORK

PG-based OLAP Systems. Neo4j [8] and TigerGraph [10] are two graph databases that adopt a native graph storage and both use SQL-like query languages. SQLGraph [55] models data as PG but adopts a relational schema in its database design. JanusGraph [3], Titan [2] and OrientDB [9] represent and store a PG in a NoSQL system. GRAKN.AI [6] and HyperGraphDB [1] are two knowledge graph systems that use PG representation for both OLTP and OLAP analytics. None of the above systems have a query execution model that naturally supports a full-fledged set of optimizations for processing online analytical queries as Grasper's Expert Model.

RDF-based Systems. The RDF model is widely used for knowledge representation. However, as RDF represents a graph as a collection of triples, which is more schematic than PG, many RDF systems store and query RDF graphs using structured tables in relational databases, such as TripleBit [69], Trinity.RDF [70] and Virtuoso [22]. Some other systems [27, 48] simply store RDF data as a massive set of triples. They adopt MapReduce to partition the RDF triples in a distributed setting and parallelize index-lookup across multiple machines for processing RDF queries. More recently, Wukong [52] and Wukong+G [58] use a KV-store to manage RDF data and leverage RDMA and GPU to process RDF queries based on graph exploration. The key differences between PG-based systems and RDF-based systems are their underlying storage and data schema, which further determine the system design and implementation for query execution and optimizations.

Other Graph Systems. A large number of systems have been proposed for batch-processing workloads in large graphs [39, 61, 66, 71]. Most of these systems [13, 26, 40, 50, 64, 65, 67, 68] follow Pregel's vertex-centric framework [40], while a few others adopt a subgraph-centric framework [17, 18, 56, 62, 63]. These systems aim at offline graph analytics and mining workloads, instead of online querying that requires low latency.

RDMA-based Systems. The benefits of RDMA stimulate the development and new designs of the high-performance distributed systems/stores, such as key-value stores [32, 41], relational databases [34, 36], and graph systems [19, 30, 60]. RDMA is also used to accelerate the performance in various areas such as in Spark [38], MPI [37], server monitoring [57]. The design considerations of using RDMA in Grasper are specific for OLAP workloads on PGs, which are different from those in above works.

8 CONCLUSIONS

We presented Grasper, a high performance system for processing online analytical queries on PGs. Grasper leverages RDMA to reduce the network cost of distributed query processing and tightly integrates the data store with the query engine for efficient communication. A distinguishing feature of Grasper is its query execution model, Expert Model, which not only enables tailored optimizations for each specific category of query steps, but also supports adaptive parallelism control and dynamic load balancing on runtime. Our experimental results show that Grasper achieves low latency and high throughput for a broad range of OLAP workloads. For future work, we have extended Grasper for OLTP workloads and the new system will be released soon.

Acknowledgments. We thank the reviewers and our shepherd Semih Salihoglu for their constructive comments and suggestions that helped improved the quality of the paper. This work was supported in part by ITF 6904945, and GRF 14208318 & 14222816.

[6]https://wiki.dbpedia.org/downloads-2016-10

REFERENCES

[1] 2010. *HyperGraphDB*. http://www.hypergraphdb.org/.

[2] 2015. *TITAN, Distributed Graph Database*. http://titan.thinkaurelius.com/.

[3] 2017. *JanusGraph, Distributed Graph Database*. http://janusgraph.org/.

[4] 2019. *Apache TinkerPop*. http://tinkerpop.apache.org/.

[5] 2019. *Cypher: the Neo4j query Language*. http://www.neo4j.org/learn/cypher.

[6] 2019. *GRAKN.AI*. https://grakn.ai/.

[7] 2019. *Gremlin*. http://tinkerpop.apache.org/gremlin.html.

[8] 2019. *Neo4j*. https://neo4j.com/.

[9] 2019. *OrientDB*. https://orientdb.com/.

[10] 2019. *TigerGraph*. https://www.tigergraph.com/.

[11] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Query Optimizations over Decentralized RDF Graphs. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 139–142. https://doi.org/10.1109/ICDE.2017.59

[12] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 451–469. https://doi.org/10.1145/2814270.2814294

[13] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[14] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. 2017. Distributed Join Algorithms on Thousands of Cores. *PVLDB* 10, 5 (2017), 517–528. https://doi.org/10.14778/3055540.3055545

[15] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine A. Yelick. 2006. Optimizing bandwidth limited problems using one-sided communication and overlap. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. https://doi.org/10.1109/IPDPS.2006.1639320

[16] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. 2009. Interactive plan hints for query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 1043–1046. https://doi.org/10.1145/1559845.1559976

[17] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[18] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 2019. Large Scale Graph Mining with G-Miner. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 1881–1884. https://doi.org/10.1145/3299869.3320219

[19] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2014. Computation and communication efficient graph processing with distributed immutable view. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*. 215–226. https://doi.org/10.1145/2600212.2600233

[20] Camille Coti, Sami Evangelista, and Laure Petrucci. 2018. One-Sided Communications for More Efficient Parallel State Space Exploration over RDMA Clusters. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. 432–446. https://doi.org/10.1007/978-3-319-96983-1_31

[21] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*. 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[22] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8. http://sites.computer.org/debull/A12mar/vicol.pdf

[23] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 619–630. https://doi.org/10.1145/2723372.2742786

[24] Zhisong Fu, Zhengwei Wu, Houyi Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, and Xi Hu. 2017. GeaBase: A High-Performance Distributed Graph Database for Industry-Scale Applications. In *Fifth International Conference on Advanced Cloud and Big Data, CBD 2017, Shanghai, China, August 13-16, 2017*. 170–175. https://doi.org/10.1109/CBD.2017.37

[25] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. 1992. Query Optimization for Parallel Execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*. 9–18. https://doi.org/10.1145/130283.130291

[26] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[27] Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB* 4, 11 (2011), 1123–1134. http://www.vldb.org/pvldb/vol4/p1123-huang.pdf

[28] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *PVLDB* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[29] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*. 60–71. https://doi.org/10.1145/1815961.1815971

[30] Kyungho Jeon, Hyuck Han, Shin Gyu Kim, Hyeonsang Eom, Heon Young Yeom, and Yongwoo Lee. 2010. Large Graph Processing Based on Remote Memory System. In *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*. 533–537. https://doi.org/10.1109/HPCC.2010.88

[31] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. 2017. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In *Handbook of Big Data Technologies*. 457–505. https://doi.org/10.1007/978-3-319-49340-4_14

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*. 295–306. https://doi.org/10.1145/2619239.2626299

[33] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue* 11, 7 (2013), 40. https://doi.org/10.1145/2508834.2513149

[34] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 355–370. https://doi.org/10.1145/2882903.2882949

[35] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable Distributed Stream Join Processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 811–825. https://doi.org/10.1145/2723372.2746485

[36] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 48–63. https://doi.org/10.1145/3064176.3064202

[37] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. 2004. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming* 32, 3 (2004), 167–198. https://doi.org/10.1023/B:IJPP.0000029272.69895.c1

[38] Xiaoyi Lu, Md. Wasi-ur-Rahman, Nusrat S. Islam, Dipti Shankar, and Dhabaleswar K. Panda. 2014. Accelerating Spark with RDMA for Big Data Processing: Early Experiences. In *22nd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2014, Mountain View, CA, USA, August 26-28, 2014*. 9–16. https://doi.org/10.1109/HOTI.2014.15

[39] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *PVLDB* 8, 3 (2014), 281–292. https://doi.org/10.14778/2735508.2735517

[40] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. 135–146. https://doi.org/10.1145/1807167.1807184

[41] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[42] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[43] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago,*

IL, USA, May 14 - 19, 2017. 12:1–12:7. https://doi.org/10.1145/3078447.3078459

[44] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. 2013. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. 255–263. https://doi.org/10.1109/BigData.2013.6691582

[45] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2006. Semantics and Complexity of SPARQL. In *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*. 30–43. https://doi.org/10.1007/11926078_3

[46] Holger Pirk, Oscar Moll, and Sam Madden. 2016. What Makes a Good Physical plan?: Experiencing Hardware-Conscious Query Optimization with Candomblé. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 2149–2152. https://doi.org/10.1145/2882903.2899410

[47] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB* 8, 12 (2015), 1442–1453. https://doi.org/10.14778/2824032.2824043

[48] Kurt Rohloff and Richard E. Schantz. 2010. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. 4. https://doi.org/10.1145/1940747.1940751

[49] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *PVLDB* 11, 4 (2017), 420–431. http://www.vldb.org/pvldb/vol11/p420-sahu.pdf

[50] Semih Salihoglu and Jennifer Widom. 2013. GPS: a graph processing system. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*. 22:1–22:12. https://doi.org/10.1145/2484838.2484843

[51] Vibhuti S. Sengar and Jayant R. Haritsa. 2003. PLASTIC: Reducing Query Optimization Overheads through Plan Recycling. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. 676. https://doi.org/10.1145/872757.872867

[52] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 317–332. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/shi

[53] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 347–353. https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi

[54] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 347–353. https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi

[55] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1887–1901. https://doi.org/10.1145/2723372.2723732

[56] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *PVLDB* 7, 3 (2013), 193–204. https://doi.org/10.14778/2732232.2732238

[57] Karthikeyan Vaidyanathan, Hyun-Wook Jin, and Dhabaleswar K. Panda. 2006. Exploiting RDMA operations for Providing Efficient Fine-Grained Resource Monitoring in Cluster-based Servers. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*. https://doi.org/10.1109/CLUSTR.2006.311916

[58] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. 651–664. https://www.usenix.org/conference/atc18/presentation/wang-siyuan

[59] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 87–104. https://doi.org/10.1145/2815400.2815419

[60] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*. 408–421. https://doi.org/10.1145/2806777.2806849

[61] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7, 1-2 (2017), 1–195. https://doi.org/10.1561/1900000056

[62] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. 2017. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR* abs/1709.03110 (2017). arXiv:1709.03110 http://arxiv.org/abs/1709.03110

[63] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB* 7, 14 (2014), 1981–1992. https://doi.org/10.14778/2733085.2733103

[64] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. 1307–1317. https://doi.org/10.1145/2736277.2741096

[65] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2018. GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit. *IEEE Trans. Parallel Distrib. Syst.* 29, 1 (2018), 99–114. https://doi.org/10.1109/TPDS.2017.2743708

[66] Da Yan, Yuanyuan Tian, and James Cheng. 2017. *Systems for Big Graph Analytics*. Springer. https://doi.org/10.1007/978-3-319-58217-7

[67] Fan Yang, Yuzhen Huang, Yunjian Zhao, Jinfeng Li, Guanxian Jiang, and James Cheng. 2017. The Best of Both Worlds: Big Data Programming with Both Productivity and Performance. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 1619–1622. https://doi.org/10.1145/3035918.3058735

[68] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *PVLDB* 9, 5 (2016), 420–431. https://doi.org/10.14778/2876473.2876477

[69] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a Fast and Compact System for Large Scale RDF Data. *PVLDB* 6, 7 (2013), 517–528. https://doi.org/10.14778/2536349.2536352

[70] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (2013), 265–276. https://doi.org/10.14778/2535570.2488333

[71] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. 2017. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. 40–51. https://doi.org/10.1145/3127479.3128606

[72] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 614–630. https://doi.org/10.1145/3132747.3132777