Pragh: Locality-preserving Graph Traversal with Split Live Migration

Paper #312

Abstract

Many real-world data like social, transportation, biology, and communication data can be efficiently modeled as a graph. Hence, graph traversal such as multi-hop or graph-walking queries has been key operations atop graph stores. However, since different graph traversals may touch different sets of data, it is impossible to have a one-size-fits-all graph partitioning algorithm that preserves access locality for graph traversal workloads. Meanwhile, prior shard-based migration faces a dilemma such that coarse-grained migration may incur more migration overhead over the increased locality benefit, while fine-grained migration usually requires excessive metadata and incurs non-trivial maintenance cost.

This paper proposes Pragh, an efficient locality-preserving live graph migration scheme for graph store in the form of key-value pairs. The key idea of Pragh is a split migration model which only migrates values physically while retains keys in the initial location. This allows fine-grained migration while avoiding the need to maintain excessive metadata. Pragh integrates an RDMA-friendly location cache from DrTM-KV to provide fully-localized accesses to migrated data and further makes a novel reuse of the cache replacement policy for lightweight monitoring. Pragh further supports evolving graphs through a check-and-forward mechanism to resolve the conflict between updates and migration of graph data. Evaluations on an 8-node RDMA-capable cluster using a state-of-the-art graph traversal benchmark shows that Pragh can increase the throughput by up to 19X and decrease the median latency by up to 95% thanks to split live migration, which eliminates 97% remote accesses. A port of split live migration to Wukong with up to 2.53X throughput improvement further confirms the effectiveness and generality of Pragh.

1 Introduction

Graph data ubiquitously exist in a wide range of application domains, including social networks, road maps, biological networks, communication networks, electronic payment, semantic webs, just to name a few examples [44]. Graph traversal (aka multi-hop or graph-walking) queries have been prevalent and important operations atop graph store to support emerging applications like fraud detection in e-commerce transaction [42], user profiling in social networking [10, 17, 5], query answering in knowledge base [49, 60], and urban monitoring in smart city [61].

With the increasing scale of data volume and the growing number of concurrent operations, running graph traversal workloads over distributed graph store becomes essential. Graph traversal workloads are severely sensitive to the access locality, while it is notoriously difficult to partition graph with good locality. For example, the difference of median latency for two-hop query (like friends-of-friends [17]) over a Graph500 dataset (RMAT26) [11] is more than 25X (0.67ms and 17.1ms) between a single machine and an 8-node cluster. Further, preserving locality is even more challenging where workloads and datasets may evolve over time, while it is common for many production applications [6, 15, 39, 32].

We argue that live migration of graph data is a necessary mechanism for preserving access locality in graph traversals, because existing alternatives have several limitations in many scenarios. First, locality-aware graph partitioning algorithms may improve the performance of a specific dataset and workload [26, 12]. However, *one partition scheme cannot fit all* [59]. Further, a proper graph partitioning scheme for a certain workload may be ineffective and even harmful to another graph traversal workload. Second, replicating data to multiple or all machines allows more (fast) localized read accesses, but also leads to excessive memory overhead as the increase of machines and heavy synchronization cost among replicas for write operations.

Hence, live migration becomes a compelling approach to preserve locality, which has been widely investigated in the database and distributed systems community over the last decade. Unfortunately, the unique characteristics of graph data and traversal operations significantly weaken the effects of live migration using a shard-based approach, even which is adopted by almost all existing systems. For example, using

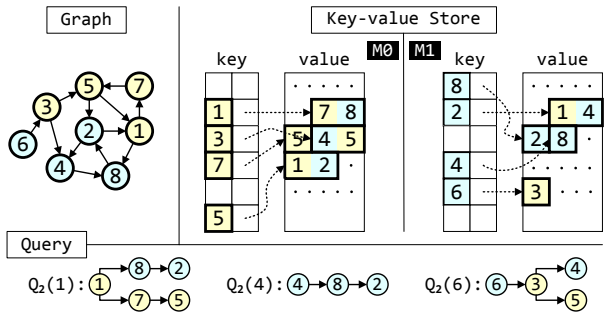

Fig. 1. A sample graph (G), key-value store over 2 machines, and three two-hop queries (Q_2).

a typical shard-based live migration [24, 57] with an optimal migration plan on above two-hop query experiment will just decrease 30% (17.1ms vs. 12.0ms) median latency, still far from the performance of ideal setting (pure localized access). Because the majority of the migrated data in a shard would likely have different location preferences. On the other hand, decreasing the size of shard (fine-grained) would incur high memory and CPU overhead due to storing and maintaining excessive metadata (a location mapping for every shard).

In this paper, we present Pragh, an efficient locality-preserving live migration scheme for distributed in-memory graph store. The key idea of Pragh is a new migration scheme called *split live migration*, which separates the migration of keys and values. Only the value would be migrated physically, while the key would always be stationary at its initial location. This allows fine-grained migration (vertex granularity) while avoiding the need to maintain excessive metadata.

Pragh is made efficient and cost-effective with several key design choices. First, to migrate well-selected vertices (scattered over the entire store) efficiently, Pragh proposes a *unilateral migration protocol* such that the target machine can migrate vertices alone by carefully leveraging one-sided RDMA primitives, while the traversal workloads can concurrently execute on the store. Second, Pragh integrates split live migrate with location-based caching [58] to provide *fully-localized* accesses to migrated data. This eliminates the restriction from the stationary key and unleashes the full power of split migration. Third, to support evolving graph with live migration, Pragh designs a *check-and-forward mechanism* to resolve the conflict between updating and migrating data. Finally, fine-grained monitoring both local and remote accesses to every vertex may incur non-trivial memory and CPU overhead to traversal workloads. Pragh makes a novel reuse of the cache replacement policy to concentrate on tracking remote data accessed frequently. Pragh further provides two optional mechanisms (eager and deferred) for local access tracking to balance the accuracy and the timeliness of migration.

We have implemented Pragh by extending DrTM-KV [58], a state-of-the-art RDMA-enable key-value store, to store graph data and support split live migration. To demonstrate the effectiveness and efficiency of Pragh, we have conducted

Table 1: A detail analysis of shard-based live migration.

	Ideal	Shard-based	
		Before	After
Throughput (K ops/sec)	2,924	120	160
Median/50th Latency (msec)	0.67	17.1	12.0
Tail/99th Latency (msec)	2.72	77.8	63.5
Remote Access Rate (%)	0	87.7	68.1
Data Migration Rate (%)	-	-	84.5

a set of experiments using a state-of-the-art graph traversal benchmark on an 8-node RDMA-capable cluster. The experimental results show that Pragh can increase the throughput by up to 19X and decrease the median latency by up to 95% through live migration, as the rate of remote accessing reduces from 87.7% to 2.8%. We also port split live migration to Wukong [49], a state-of-the-art graph-based RDF store that provides low latency and high throughput for concurrent SPARQL query processing. An evaluation using original concurrent workload benchmark [49] shows that the throughput increases by 2.53X due to using split live migration.

2 Background and Motivation

2.1 Graph Store and Traversal Workload

The graph-structured store (aka graph store) becomes more and more prevalent in an increasing number of applications [44] for modeling the relationships among connected data. Due to fast lookup and good scalability, distributed key-value stores are widely used by existing graph systems [49, 61, 54, 21, 30, 23, 48, 60, 51] as the underlying storage layer to support graph traversal operations efficiently, which play a vital role for many emerging and crucial applications [42, 10, 17, 60, 49].

A natural way to build a graph model on top of the key-value store is to simply use the vertex as the key and the adjacent list as the value [48]. Further, separate key and value memory regions are used to support variable-sized key-value pairs [36, 58, 49]. Specifically, the key region is a fix-sized hash table, where each entry stores a key and an address (i.e., offset and size) of the value region. The value region stores variable-sized values consecutively. As shown in Fig. 1, a sample graph (G) is stored into a key-value store over two machines. Various graph traversal operations (like FoF, multi-hop query, and random walking) can be implemented by iteratively accessing key and value pairs. For example, the two-hop query on vertex 1 ($Q_2(1)$) will first retrieve neighbors of the start point (vertex 1) by hashing it as the key and accessing its value (vertex 7 and 8). The next hop will use the value in this hop as the keys (hash(7) and hash(8)) to retrieve their neighbors (vertex 2 and 5) recursively. The accesses over key and value may be either local or remote according to the partitioning scheme.

2.2 Poor Locality and Partitioning

For distributed in-memory stores, the locality of data accessing is quite important because accessing local memory is still more than 20× faster than accessing remote memory across networks, even using high-speed networks [21]. Unfortunately, the traversal on distributed graph data is notoriously slow due to poor data locality. Prior work shows that assigning vertices to N machines randomly will lead to the expected fraction of remote accesses reaching $1 - \frac{1}{N}$ [26].

To illustrate the performance impact of locality for graph store, we conducted a motivating experiment using two-hop queries (like FoF [17]) over Graph500 dataset [11] (RMAT26) on an 8-node RDMA-capable cluster. The graph is partitioned into 8 machines randomly (hash-based), and a set of vertices randomly sampled with a Zipf distribution (θ=0.99) is used to run two-hop queries which access fixed 100 friends of 100 friends. As shown in Tab. 1, the distributed setting using 8 machines only achieves less than 4% throughput (120 vs. 2,924) and more than 25× median (50^th percentile) latency (17.1 vs. 0.67) of the ideal setting since the rate of remote accessing reaches up to 87.7%.[1]

Therefore, designing locality-aware graph partitioning algorithms has been an active area of research for a decade [26, 12], especially for graph analytics systems. However, *one partition scheme cannot fit all* [59]. It is hard or even impossible to handle dynamic workloads or evolving graphs only relying on static partition-based approaches. One example is shown in Fig. 1 such that $Q_2(1)$ and $Q_2(4)$ contends for the same vertices (8 and 2). The queries may arrive at different times which causes *false contention*. Actually, prior work on production applications has shown that workloads change rapidly over time [6, 15, 39, 32].

2.3 Live Migration

Live migration (aka dynamic migration) is a compelling approach for handling dynamic workloads and has been widely investigated in the database and distributed systems communities [19, 20, 25, 24, 34, 57, 4]. Generally, a centralized coordinator will make the migration plan according to the statistics (e.g., access frequency) collected by the monitor on each machine. The migration threads on source and/or target machines will implement the plan by migrating key-value pairs in a synchronous way (see Fig. 2(b)). Since the position of vertices may change after migration, additional metadata (POS) will be accessed to lookup the latest positions of the key-value pairs before accessing them (see Fig. 2(a)). The metadata should be updated by the coordinator during live migration and usually is consistently cached at each machine to avoid remote lookup for every accesses.

[1]The ideal result is gained by running the benchmark on a single machine (fully local accessing). The throughput is magnified 8× (the number of machines).

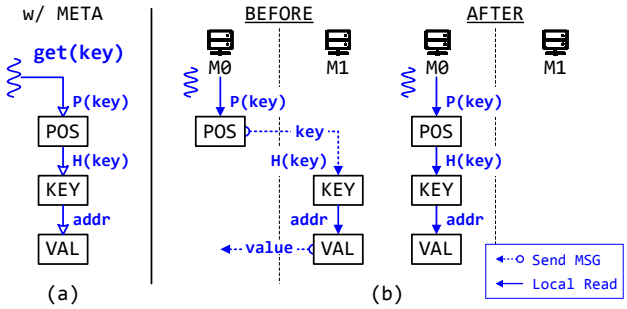

Fig. 2. (a) The sequence of an access on (kv-based) graph store and (b) a comparison of accesses before and after live migration.

Shard-based migration. A ubiquitous approach in live migration is to group the data into *shards* (*partitions*) by key ranges or key hashing [46, 50, 3, 34, 4]. Shards serve as the unit of migration for load balancing and locality-aware optimization. Prior work mainly focuses on relational workloads (e.g., TPC-C) or simple CRUD workloads (e.g., YCSB [14]). Compared to traversing graph data, such workloads with datasets usually have high access locality (e.g., accessing 1% remote key in TPC-C). Consequently, leveraging shard-based migration on the graph store is ineffective and may be harmful, due to the following reasons:

First, migrating data at shard granularity will significantly weaken the effects of data migration. Due to lacks of data locality, the majority of the migrated data would likely not be accessed by the workload at the target machine. Meanwhile, it will also increase the number of remote accesses at the source machine. Based on the above motivating experiment, we partition graph data into one hundred shards per machine, similar to prior work [10, 4]. All of local and remote accesses to every shard are monitored and aggregated to make an optimal migration plan. As shown in Tab. 1, the rate of remote accessing only decrease from 87.7% to 68.1% even after migrating more than 84.5% of graph data (about 20GB). As a result, the throughput only increases 37% (120 vs. 160) and the median latency also just decreases 30% (17.1 vs. 12.0), still far from the performance of an ideal setting.

Second, though decreasing the size of shard could enhance the effectiveness of migration, it still faces the same drawbacks of static graph partitioning approaches when handling dynamic workloads, unless vertices (key-value pairs) serve as the unit of migration. For example, two irrelevant queries may contend the same shard even assigning two vertices to one shard by key ranges, like vertex 2 and 4 for $Q_2(1)$ and $Q_2(6)$ in Fig. 1. More importantly, the amount of metadata (POS) needed to manage the shards would incur extremely high memory pressure. For example, the metadata for the motivating experiment will consume about 3G memory on each machine to support vertex granularity migration. Each machine has to cache the entire metadata since the workload may access any vertex of the graph. Consequently, the metadata may exceed the graph data when the graph scales.

3 Approach and Overview

Our approach: split live migration. We propose a new migration approach, named *split live migration*, that enables live migration at the minimum level of granularity (i.e., key-value pair). A landmark difference compared to prior approaches is that *split live migration has no need of metadata at all*. This is the greatest advantage but also the biggest challenges for live migration.

The key principle of split migration is to separate the migration of keys and values. The key will always be *stationary* at its initial location, which can be found without metadata (e.g., key hashing). The value will be *migratory* on demand to improve locality or rebalance the load. Our design naturally tackles the issue of memory pressure by avoiding metadata due to the stationary key. Further, allowing fine-grained migration (even a single value) would maximize the effectiveness of data migration for graph store. However, there are still many challenges before making split live migration come true.

Opportunity: RDMA. Remote Direct Memory Access (RDMA) is a networking feature to provide cross-machine accesses with high speed, low latency, and kernel bypassing. Much prior work has demonstrated the benefit of using RDMA for in-memory key-value stores [36, 21, 31, 58]. Generally, the get/put (read/write) operation first uses RDMA READs to lookup the location (address) of the value by hashing the given key, and then use RDMA READ/WRITE to retrieve/update the value (see the left part of Fig. 4(b)). We observe that *RDMA one-sided primitives (READ, WRITE, and CAS) decouple the accesses of keys and values, which make it easy and efficient to separate keys and values in physical.* It opens a new opportunity to split live migration

Challenges and solutions. First, split live migration uses the key-value pair as the unit of migration, such that the key-value pairs which will be migrated are scattered over the entire graph store. Therefore, directly using existing protocols designed for shard-based migration may be inefficient. We propose a unilateral (target-only) migration protocol that the target machine can do it alone and efficiently by carefully leveraging one-sided RDMA primitives (§4.1).

Second, the basic split migration only migrates the values of key-value pairs, which can at most eliminate about half of the remote accesses. This is because the read access to the key of key-value pair (lookup the location of the value) will still be remote. We address this challenge by integrating split migration with RDMA-friendly location-based caching [58] to provide *fully-localized* access to migrated data (§4.2).

Third, the split of key and value after performing migration presents a new challenge to the support of evolving graph, especially for the target-only protocol. We use a check-and-forward mechanism to resolve the conflict between data updating and data migrating tasks (§4.3).

Finally, to maximize the effectiveness of data migration,

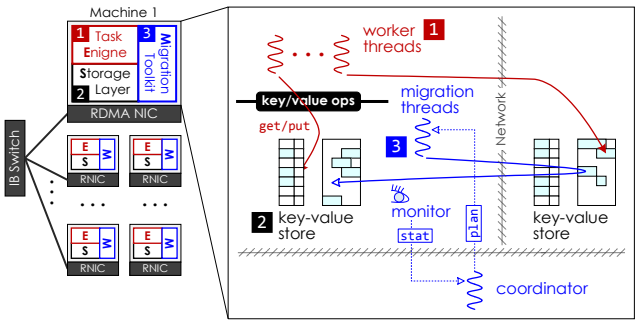

Fig. 3. The architecture of Pragh.

both local and remote accesses to every key-value pair should be tracked to generate an optimal migration plan. It may incur non-trivial memory and CPU overhead to traversal workloads. We design a lightweight, memory-saving monitor, which reuses the location cache to track frequent remote accesses and provides two optional mechanisms for local access tracking to balance the accuracy and the timeliness of live migration (§4.4).

Architecture. As shown in Fig. 3, Pragh is a distributed in-memory graph store with split live migration. It follows a decentralized architecture to deploy servers on a cluster of machines connected with a high-speed, low-latency RDMA network. Each server is composed of three components: task engine, a storage layer, and migration toolkit. The task engine binds a worker thread on each core with a task queue to continuously execute operations (e.g., get and put) from clients or other servers. The storage layer adopts an RDMA-enabled key-value store over distributed hashtable to support a partitioned global address space. The migration toolkit enables a monitor to collect statistics of graph store and runs migration threads to perform live migration. Pragh scales by partitioning graph data randomly (hash-based) into multiple servers. Each server stores a partition of the graph, which is shared by all of the workers and migration threads on the same machine.

Execution flow. Pragh is designed to handle concurrent operations on graph data with low-latency and high-throughput. The key advantage of Pragh over previous systems is capable of physically migrating data to improve locality in a split way, which can promptly and significantly enhance performance for dynamic workloads.

Similar to prior work [50, 57, 34], a centralized coordinator will make migration plan according to the statistics (e.g., access frequency) collected by the monitor on each server and migration policies. The details – how to make a proper policy and how to find an optimal plan – are beyond the scope of this paper and are part of our future work. On each server, the monitor will periodically track the accesses of worker threads to the key-value store in the background and report to the coordinator on demand. For example, when the number of accesses to a remote key-value pair exceeds

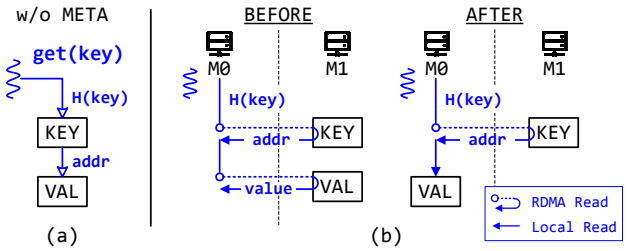

Fig. 4. (a) The sequence of an access on (kv-based) graph store without meta-data and (b) a comparison of accesses before and after split live migration.

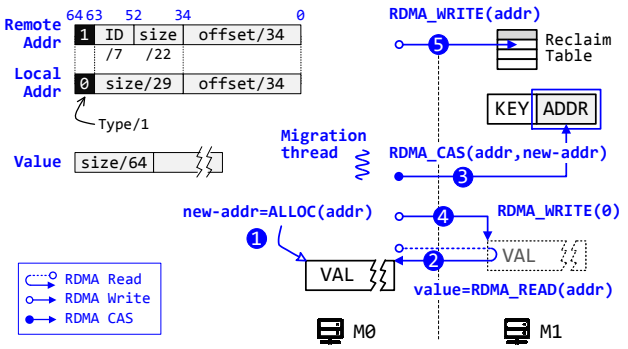

Fig. 5. The execution flow of basic split migration.

a threshold. The migration threads will move the key-values pair physically between servers according to the plan generated by the coordinator, while the worker threads will continue to execute operations by accessing the same key-value store concurrently.

4 Split Live Migration

Pragh uses an RDMA-enabled key-value store over distributed hashtable to store graph data physically. For brevity, Pragh supposes that each vertex has a unique ID (vid) and use it as the key. The hash value of the key (`H(key)`) can be used to identify the host server and the location in the key region. As shown in Fig. 4(a), to get neighbors of a given vertex, the worker thread first uses `H(key)` to lookup the address of its value and then retrieves the value (a list of IDs of neighbors). For remote key-value pairs, RDMA READs are used to access keys and values (see the left part of Fig. 4(b)), which are at least 20× slower than local reads. Hence, Pragh uses split live migration to eliminate such remote accesses.

The following section will provide a detailed description of split live migration employed by Pragh.

4.1 Basic Split Migration

We start from the basic migration protocol, assuming that there only exist traversal workloads (i.e., get operations) in the graph store. Since the key is always stationary in the split migration, Pragh will only move the value to the target machine. This could improve locality by avoiding remote accesses to the values (see the right part of Fig. 4(b)).

Address layout. To avoid the influence between accessing and migrating key-value pairs, the address (within the key) should be changed from local to remote in a lock-free way (e.g., compare-and-swap). Therefore, both the local and remote location of value should use a 64-bit address uniformly, which can be modified atomically using both local and RDMA atomic instructions.

Considering the machine ID should be added into the address, a simple layout may severely restrict the scope of address space. Pragh adopts a differentiated layout for local and remote addresses. As shown in the top left corner of Fig. 5, The most significant bit is used to present the type of address, local (0) or remote (1). For local addresses, the rest of the bits are used to store 29-bit value size and 34-bit offset within the value region. Thus, the size of value region and a single value on a single machine can reach 128GB and 4GB respectively (assuming 8-byte granularity and alignment). For remote addresses, the value offset still occupy 34 bits to present the entire remote value region, while the value size reduces to 22 bits for hosting 7-bit machine ID. Thus, the graph store can scale up to 128 machines, while the size of maximum value that can be migrated is limited to 32MB. The observation is that the system will prefer to migrate the workloads rather than very large key-value pairs. Further, a large key-value pair can be split into multiple ones (vertex decomposition [49]), and each one can be migrated separately.

Unilateral migration protocol. Similar to traditional shard-based migration systems, the split live migration also could be implemented by the collaboration of migration threads on source and target machines. However, the key-value pairs which will be migrated, are scattered over the entire graph store due to lacks of locality. It means that migrating multiple key-value pairs may incur a prolonged interruption to the concurrent graph accessing and/or lengthy migration delay since multiple addresses (within separated keys) should be modified by atomic operations (e.g., compare-and-swap).

Pragh proposes a unilateral migration protocol based on one-sided RDMA primitives. It only uses the migration thread on the target machine to migrate the key-value pair instantly, while the worker threads on every machine can still access the key-value pair concurrently. Fig. 5 illustrates three steps of the migration protocol (a detail pseudo-code is shown in Fig. 6). First, the migration thread on target machine will allocate a memory space in local value region (`new_addr`) to host migrated value (❶), according to the size in the original address (assuming it has known). Second, the migration thread will retrieve the value using one RDMA READ from the original address to new address (❷). Finally, one RDMA CAS is used to replace the original (local) address with the new (remote) address (❸).

Invalidation and reclaim. Unilateral migration protocol will incur two new problems. First, the memory of migrated

```
   MIGRATE(key)
   1 retry:
   2   kmid = H(key)              ▸ e.g., key % machines
   3   addr = LOOKUP(kmid, key)
❶  4   buf = ALLOC(addr.sz)
   5   new_addr = {1, local_mid, buf, addr.sz}
❷  6   RDMA_READ(addr.mid, buf, addr.off, addr.sz)
❸  7   if !RDMA_CAS(kmid, hash(key), addr, new_addr)
   8   | goto retry                ▸ conflict w/ put
   9   zero = 0                   ▸ invalidate value
❹ 10   RDMA_WRITE(addr.mid, &zero, addr.off, 8)
❺ 11   RDMA_WRITE(addr.mid, addr, reclaim, 8)  ▸ reclaim
```

Fig. 6. Pseudo-code of MIGRATE operation.

value in source machine should be invalidated. However, some worker threads may still have the original address of the migrated value and access it in future. To solve it, Pragh proposes a passive invalidation mechanism. The migration thread will invalidate the original memory of migrated value by zeroing (RDMA WRITE) the size within the value (❹). Before using retrieved value, the worker thread should check whether the size within the value and address are equal. If not, the address should be regained. Note that the worker thread can safely read the value from original memory before invalidation even it has just been migrated (❸).

Second, the memory of migrated value in source machine should be reclaimed. However, it is hard or even impossible for the migration thread on the target machine to solely free the memory on the source machine. Therefore, Pragh uses a lease-based mechanism to reclaim the memory of migrated values in background by a garbage collection (GC) thread on source machine.[2] The migration thread will actively write (RDMA WRITE) the original address to the reclaim table[3] of source machine at the end of live migration (❺). The GC thread on each machine will periodically check the reclaim table to free expired memory, which has been migrated before a pre-agreed lease. All of the worker threads comply with the convention that the value address obtained before the lease should not be used. The performance impact could be trivial by using a long-term lease.

4.2 Fully-localized Split Migration

The basic split migration only avoids remote accesses to the values, which limits the effects of migration since only at most half of remote accesses can be eliminated.

Observation: location cache. Prior work [58, 22, 56] proposes location-based caching for RDMA-friendly key-value stores, which aims at avoiding remote accesses to the keys. Different to caching the content (value) of key-value pairs, the location cache (L$) only stores the location (address) of key-value pairs, which is very space-efficient and effective (see the left part of Fig. 7). We observe that *location cache is a perfect counterpart to split migration*. They focus on two different halves of the access to the remote key-value pair, and the candidates of them are also well matched, namely remote key-value pairs frequently accessed. Finally, a small cache has negligible memory overhead (e.g., 128MB) and lookup cost, yet it is sufficient to achieve fully-localized accesses for most workloads [43, 58].

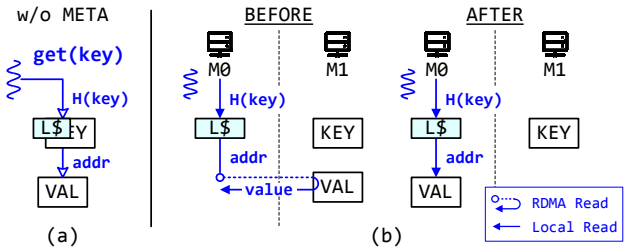

Fig. 7. (a) The sequence of an access on (kv-based) graph store with location cache and (b) a comparison of accesses before and after split live migration with location cache (for remote kv pair).

Integration with location cache. Pragh extends the graph store with location cache (L$) and integrates it with split live migration to enable fully localized accesses after migration. Fig. 8 illustrates the pseudo-code of get operation with the integration of location cache and split live migration. When accessing a remote key-value pair (Line 9), the worker thread will first check location cache (Line 20) and fill the cache (if missed) with the address of the value (Line 24) obtained by the remote access to the key (Line 23). Given the address, the worker thread will retrieve the value using one RDMA READ (Line 15).

If the worker threads access the key-value pair frequently enough, the value will be migrated to the local using the basic migration protocol. After that, the address stored in location cache will be updated by the new address, which points to the local value region. Therefore, the accesses to the key-value pair will be fully localized (Line 11-13), as shown in the right part of Fig. 7. On the other hand, the local key-value pair could also be migrated to other machines, thus the type of address will be used to decide how to retrieve the value (Line 5-8).

Finally, the address stored in the location cache should also follow the convention of the invalidation and the reclaim mechanism. First, if retrieved value is invalid (Line 16), the worker thread has to delete the address in location cache for the remote key-value pair (Line 17 and 18), and needs to retry (Line 19). Second, the cached address must expire after a lease from the last cache time (Line 21 and 25).

4.3 Full-fledged Split Migration

The basic migration protocol only considers traversal workloads (i.e., get operations) concurrently execute in the graph store. Pragh extends it with a check-and-forward mechanism to support evolving graph (i.e., put operations). For brevity, suppose that graph store has provided some mech-

[2]Pragh uses the precision time protocol (PTP) [1] to implement lease.
[3]Pragh implements the reclaim table like the circular buffer in prior work [21].

```
 GET(key, buf)
+1 retry:
 2   kmid = H(key)            ▸ e.g., key % machines
 3   if kmid == local_mid                ▸ local key
 4   | addr = keys[key]
+5   | if addr.type == 0                ▸ local value
 6   | | MEMCPY(buf, vals[addr.off], addr.sz)
+7   | else                  ▸ remote value (migrated)
+8   | | RDMA_READ(addr.mid, buf, addr.off, addr.sz)
 9   else                                ▸ remote key
 10  | addr = LOOKUP(kmid, key)
+11  | if addr.type == 1                  ▸ migrated
+12  | |  && addr.mid == local_mid      ▸ local value
+13  | | MEMCPY(buf, vals[addr.off], addr.sz)
+14  | else                             ▸ remote value
 15  | | RDMA_READ(addr.mid, buf, addr.off, addr.sz)
+16  if CHECK(addr, buf)
+17  | if kmid != local_mid
+18  | | cache.DELETE(key)               ▸ invalidate
+19  | goto retry

 LOOKUP(mid, key)
x20  if cache.FIND(key)
+21  |  && !EXPIRED(cache[key].lease)
x22  | return cache[key]                 ▸ cache hit
 23  RDMA_READ(mid, addr, hash(key), 8)
x24  cache.INSERT(key, addr)            ▸ fill cache
+25  cache[key].lease = NOW()
 26  return addr
```

Fig. 8. Pseudo-code of GET operation with location cache. The code lines with "x" and "+" stand for additional instructions to integrate with location cache and split live migration, respectively.

anisms [49, 61] to run traversal workloads over evolving graph correctly, thus Pragh only tackles the conflict between split live migration and the change of graph. More specifically, Pragh only needs to consider the concurrent update to edges (i.e., change the value of a key-value pair).

We observe that both migrate and put operations will change the address within the key atomically to mark the success of processing (Line 7 in Fig. 6 and Line 6 in Fig. 9).[4] Moreover, the put operation will always be assigned to the machine hosting the key at first. So for key-value pairs migrated, a better choice is to forward the put operation to the machine hosting the value upon conflicts, which also ensures consistency. Therefore, Pragh adopts different strategies for migrate and put operations when detecting the conflict over the address; migrate operation will retry (Line 8 in Fig. 6), while put operation will forward itself (Line 4 in Fig. 9).

4.4 Lightweight Monitoring

To generate a proper migration plan, the coordinator should collect the statistics of both local and remote accesses to every key-value pair. A (much) higher remote access number from a certain machine to a key-value pair in the most recent interval (e.g., 10s) indicates that migrating the key-value

[4]Suppose that put operation will change the size or the offset of the address (or both), namely `addr` is not equal to `new_addr`.

```
 PUT(key, val)
 1 retry:
 2   addr = keys[key]
+3   if addr.mid != local_mid
+4   | SEND(addr.mid, key, val)          ▸ forward
 5   new_addr = WRITE_VALUE(addr, val)
 6   if !CAS(hash(key), addr, new_addr)
 7   | goto retry     ▸ conflict w/ put or migrate
```

Fig. 9. Pseudo-code of PUT operation. The code lines with "+" stands for additional instructions to support split live migration.

pair to that machine may improve locality (fewer remote accesses). It has been a great challenge to track the accesses at the granularity of key-value pair.[5] Even worse, the remote accesses using RDMA READ contributes much more extra burdens (both memory and CPU overhead) to the monitor, since each machine has to track the accesses to remote key-value pairs (except local key-value pairs).

Pragh designs a lightweight, memory-saving monitor for split live migration by tracking local and remote accesses separately. For remote accesses, worker threads may access any key-value pairs, while the monitor may (very likely) only care about remote key-value pairs *accessed frequently*. This observation also matches the intention of the location cache. Hence, Pragh reuses the cache to track (partial) remote accesses. The monitor relies on the replacement policy of cache to recognize the key-value pairs (worth tracking) freely. Note that the migrated key-value pairs will still be tracked even being accessed locally (through cache).

For local accesses, reserving space for every key and tracking every access might be not worth, especially for a very large store. This is because only a small fraction of key-value pairs should be migrated for a while. For example, migrating less than 0.2% of key-value pairs is sufficient for the motivating experiment (§6.2). Therefore, Pragh allows to skip tracking local accesses and provides two optional mechanisms to balance the timeliness and the accuracy of live migration. Suppose that the remote access frequency to a key-value pair from a certain machine exceeds a threshold, that machine will report it to the coordinator.

Eager migration: The coordinator will eagerly approve the migration of the key-value pair. After migration, the machine hosting the key will track the (remote) accesses to the key-value pair using an individual table, and then may migrate it back in future if it accesses the key-value pair more frequently.

Deferred migration: The coordinator will notify the machine hosting the key to track the (local) accesses to the key-value pair using an individual table. After a migration interval, the coordinator will decide whether to migrate the key-value pair according to the statistics from all of the machines.

[5]Relational database can leverage table schema to reduce the number of tuples should be tracked, by grouping co-accessed tuples into blocks [50, 24, 47, 57]. Unfortunately, graph store are generally schema-less.

4.5 Discussion

Even though split live migration is currently implemented using RDMA, we believe that it can still benefit graph traversal workloads without RDMA, including no need of metadata and vertex granularity migration. However, after migrating the value to local, the cost to retrieve the address would be almost the same as the cost to retrieve the value directly. Hence, location cache must be deployed even without RDMA. On the other than, the lack of RDMA would also need to rethink the implementation of migration protocol. Our future work may extend Pragh to support non-RDMA network.

5 Implementation

Fault tolerance. Pragh supposes distributed in-memory graph store has provided durability and/or availability by using specific mechanisms like checkpointing or replication. Pragh only needs to consider the interrupted migration tasks and the recovery of crashed machines, because split live migration only change the location of key-value pairs rather than the content of key-value pairs.

Interrupted migration tasks: If the crashed machine is the source of migration, there is nothing to do since the key-value pair will be recovered on the crashed machine later. If the crashed machine is the target of migration, a corner case that the interruption occurs after replacing address (❸ in Fig. 5) but before reclaiming memory (❺ in Fig. 5) will cause a little memory leakage, which can be detected and reclaimed by scanning the entire value memory region in background.

System recovery: Pragh relies on the mechanism provided by graph store to detect machine failures, like Zookeeper [29]. It will notify surviving machines to assist the recovery of crashed machines, which needs to handle two kinds of key-value pairs. First, the key-value pairs hosted by a crashed machine will be reloaded by the substitute of the crashed machine. Before that, all surviving machines will flush addresses in location cache which point to the key-value pairs hosted by crashed machines (i.e., `H(key)`) whether they have been migrated or not, and reclaim the memory of values migrated from crashed machines. Second, the key-value pairs, hosted by a surviving machine but migrated to a crashed machine, will be reloaded by the surviving machine. Before that, all surviving machines will also flush addresses in location cache which point to the key-value pairs migrated to the crashed machines (i.e., `addr.mid`). Moreover, all workloads running on surviving machines involving crash machines will be aborted and suspended until recovery is complete. Finally, the coordinator in Pragh is stateless and easy to recover. The coordinator failure will not influence the execution of worker threads and only pause launching new migration tasks. The migration thread can continue to complete the outstanding migrations.

Optimizations. Pragh migrates one key-value pair (vertex) at a time by using five RDMA one-sided operations: two READs to lookup and retrieve the value, one CAS to change the address atomically, and two WRITEs to invalidate and reclaim the original memory. Though this approach can provide instant response to migration demands and fully bypass the CPU and kernel of source machine, the throughput of migration may be bottlenecked by the network due to too many RDMA operations with small payloads.

To remedy it, Pragh enables three optimizations to further accelerate split migration. First, in most cases, the migrated key-value pair is frequently accessed by the target machine, thus its address (very likely) has been already cached in location cache. It means that the migration thread can skip the first RDMA READ to lookup the address. Second, Pragh will migrate multiple key-value pairs concurrently in a pipelined fashion to better utilize network bandwidth. Each RDMA operation to migrate one key-value pair is implemented as one stage, and Pragh schedules these stages without waiting for the request completion. Finally, since the memory invalidation and reclaim are not on the critical path to migrate one key-value pair[6], Pragh enables passive ACK [56] to acknowledge the completion of such two RDMA (WRITE) requests passively, which reduces the network bandwidth. As a result, a single migration thread is sufficient to migrate more than one million vertices per second (§6).

Load balance. Though Pragh mainly focuses on using live migration to improve the locality of graph traversal workloads, it also can be used to rebalance load across machines, similar to prior work [50, 57, 34]. Basically it all depends on the migration plan generated by the coordinator. Generally, the traversal workload will be sent to the machine hosting the initial vertex and run to completion. The remote key-value pairs will be retrieved by RDMA operations. Therefore, the coordinator should recognize such hotspots and generate proper plans to scatter them over all of the machines using live migration, like Pragh. Meanwhile, different goals also need different migration policies and statistics. It is orthogonal to the design of Pragh and beyond the scope of this paper.

6 Evaluations

6.1 Experimental Setup

Hardware configuration. All evaluations were conducted on a rack-scale cluster with 8 nodes. Each node has two 12-core Intel Xeon E5-2650 v4 processors and 128GB DRAM. Each node is equipped with two ConnectX-4 MCX455A 100Gbps InfiniBand NIC via PCIe 3.0 x16 connected to a Mellanox SB7890 100Gbps IB Switch, and an Intel X540 10GbE NIC connected to a Force10 S4810P 10GbE Switch. In all experiments, we reserve four cores on each CPU to

[6]To ensure consistency, the (original) memory invalidation must be completed before the next `put` operation on (new) memory starts, which is easy to implement with the check-and-forward mechanism (§4.3).

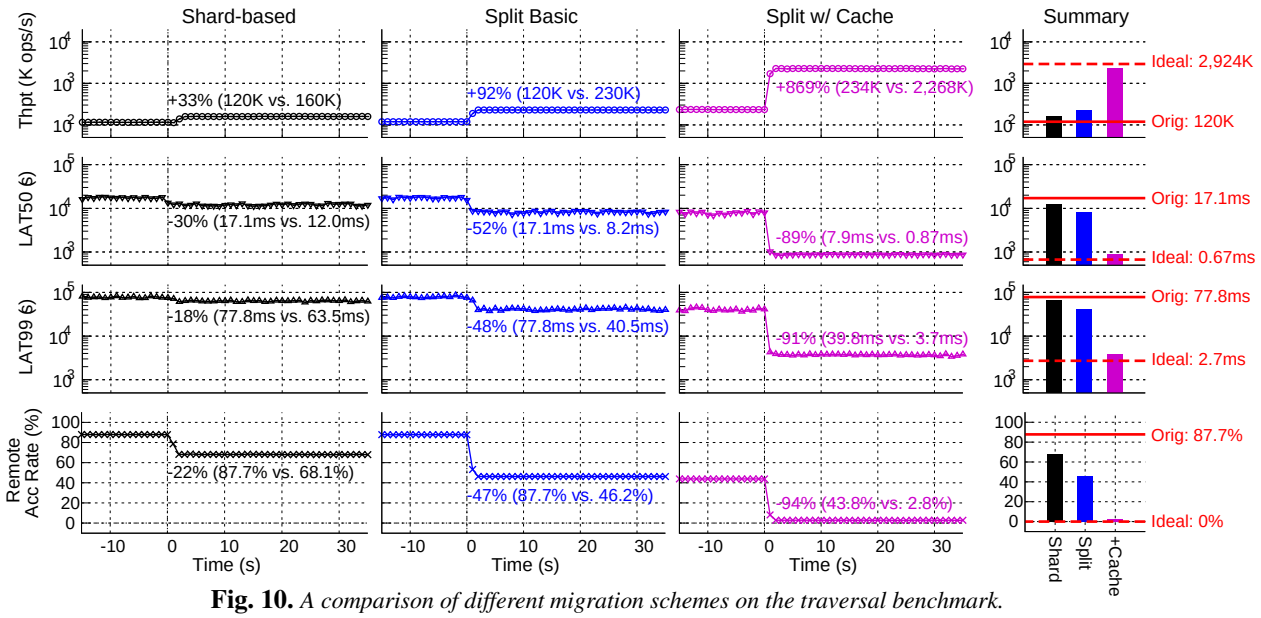

Fig. 10. *A comparison of different migration schemes on the traversal benchmark.*

generate requests to avoid the impact of networking between clients and servers as done in prior work [53, 55, 58, 13, 57, 49]. All experimental results are the average of five runs.

Traversal benchmark. Inspired by YCSB [14], we build a simple benchmark to evaluate the effectiveness of different migration approaches for graph traversal workloads. The traversal benchmark uses a synthetic graph provided by Graph500 [11]. In this paper, the graph with 2^{26} vertices and 2^{30} edges (RMAT26) is used as default dataset since we need to run the benchmark on a single machine to gain the performance of ideal setting (pure localized access). Note that the experimental results on larger graphs (e.g., RMAT29) is similar. The traversal benchmark only consists of two-hop queries and edge updates/inserts, like the *reads* and the *writes/inserts* operations in YCSB [14]. Since the majority of many traversal workloads [10] are two-hop queries and it is easy to compose other complicated queries like SPARQL query [49]. The starting vertices of two-hop queries are chosen according to a Zipf distribution with $\theta = 0.99$. The scope of starting vertices and the number of neighboring vertices retrieved could be configured. The default values are 1000 and 100, respectively. We will compare the performance impact with different settings in separate experiments.

Comparing targets. The following five results are provided in the evaluation of the traversal benchmark. ***Orig*** indicates the performance of running the benchmark over the graph data partitioned randomly and without data migration. ***Ideal*** is the result gained by running the benchmark on a single machine. Specifically, throughput is simple magnified by the number of machines (i.e., 8×). ***Shard-based*** represents the performance of a shard-based migration approach, which deploys one hundred shards at each machine, similar to prior work [10, 4]. Note that we always generate optimal migration plans for shard-based migration by tracking every access but do not consider the tracking cost. ***Split*** and ***Split/Cache*** are the performance of Pragh using split live migration with and without location cache. The size of the location cache is 128MB. The migration plan is built by the statistics collected by our lightweight monitor. The default interval is set to 10 seconds and eager migration is used as default.

6.2 Migration Effect

To study the effects of migration approaches, we run the traversal benchmark using different migration approaches and compare to the result of the original and ideal settings. As shown in Fig. 10, the original throughput and latency is about 25X slower than the ideal results (120K vs. 2,924K) since about 87.7% accesses to the key-value store is remote. Shard-based approach can only increase the throughput by 33% (120K vs. 160K) and decrease the median (50^{th} percentile) latency by 30% (17.1ms vs. 12.0ms) as it just remove about 22% remote accesses. Pragh can almost double the throughput and reduce the latency by half, thanks to the basic split migration, which removes almost all remote accesses to the values. By enabling location cache, Pragh removes almost all remote accesses to the keys and achieves a similar performance improvement. The cache hit rate is about 99%. When combining two techniques, the throughput of Split/Cache can further increase by 9.7× (19× compare to original), reaching 2,268K queries per second. It has achieved close to 80% of ideal performance. The remaining 2.8% of remote accesses is due to the competition on vertices shared by multiple queries running on different machines. Note that traditional migration scheme is hard to integrate with location cache, since they will migrate both keys and values physically and make location cache useless.

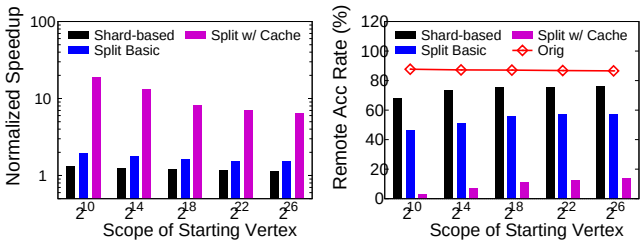

Fig. 11. *A comparison of migration effects for different approaches with the increase of scopes of starting vertices.*

Migration time and network traffic. Both split migration and shard-based migration can complete migration in seconds since we optimize the data transmissions in both methods. For shard-based migration, we migrate the shards in block granularity to fully utilize network bandwidth. However, split migration is still faster even using more network round-trips for one key. This is because fine-grained migration migrates much less data than coarse-grained, shard-based migration. For this experiment, only 77077 keys (0.13% of the total vertices) are migrated in split migration, where 730MB data is migrated in total. For comparison, 84.5% of shards (676 out of 800) are migrated in shard-based migration with a total size of 20GB data to transfer.

Scope of starting vertices. Generally, the query will start from a certain type of vertices (e.g., users or tweets in social networks), the size of subset of vertices may be various. Fig. 11 further presents the impact of using different scopes of starting vertices in the traversal benchmark from 2^{10} to 2^{26}. The speedup after migration decreases with the increase of scope sizes steadily, since the increase of possibility of contention on key-value pairs accessed by multiple queries. It will also result in the increase of remote accesses (see Fig. 11(b)).

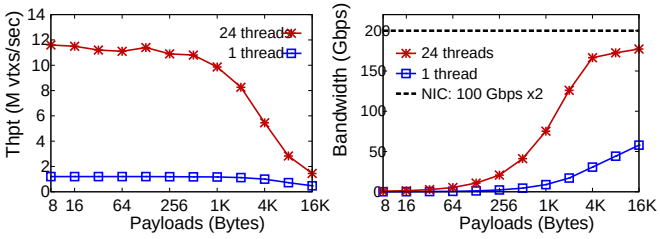

Fig. 12. *The throughput and bandwidth of unilateral migration using 1 and 24 threads.*

6.3 Migration Speed

To evaluation the capability of unilateral migration protocol, we conduct an experiment to migrate values from remote machine to local with full speed. Fig. 12 shows the throughput of migration and network bandwidth consumed with the increase of payload (i.e., value) size. A single thread is enough to migrate values for millions of vertices per second with less than 4KB payloads. Using parallel migration with 24 threads

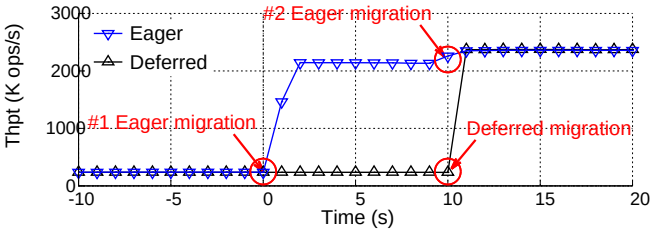

Fig. 13. *A comparison of split migration (w/ cache) with eager or deferred mechanism.*

can further increase the throughput of moving values to more than 10 millions per second. Further, using multiple RDMA primitives to migrate a single value will not be limited by network. It should be noted that split live migration will only use the CPU of target machine.

6.4 Eager Migration vs. Deferred Migration

Pragh provides two optional migration mechanisms, eager and deferred, to balance the accuracy and the timeliness of live migration. Fig. 13 compares these two mechanism using the traversal benchmark. The monitor on each machine tracks remote accesses and reports the statistics to the coordinator periodically. After receiving statistics at 0 second, the coordinator adopts different mechanisms to notify migration threads. For eager migration, all of migration threads will start to migration immediately, and the throughout reflects the effects of migration immediately, increasing from to 239K to 2,142K. However, since the migration plan may not the optimal, the second migration happens at the next interval (after about 10s). The throughput further increase to 2,362K. For deferred migration, the coordinator will only ask monitors to track the local accesses on the potential key-value pairs for migration at 0 second, and do the migration with optimal plan at the next interval. The throughput will directly increase from 239K to about 2,362K.

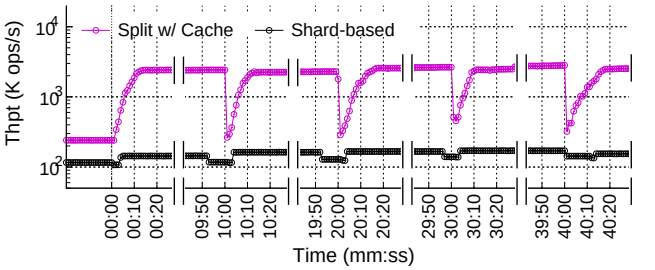

Fig. 14. *XX.*

6.5 Dynamic Workloads

to study the effectiveness of split live migration in the face of dynamic workloads, we change workloads every 10 minutes by using non-overlapping scopes of starting vertices. As shown in Fig. 14, the performance notably drops every time the workloads change. Because the location of vertex migrated for the current workload is very likely not suitable for the next workload. Shard-based migration can only provide

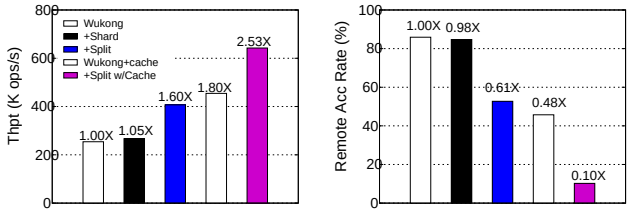

Fig. 15. *Wukong (a) throughput comparison and (b), remote access rate under skewed workloads.*

very limitation performance improvement as expected. Split migration with location cache can recover the performance after migration. Note that Pragh uses instant migration in this case, which is hard to implement in traditional migration approaches. When the monitor detect the frequency of accesses to some remote key-value pair, it will immediately report to coordinator. The migration on every machine can move values at anytime, and there is no need to synchronize with other machines.

6.6 Application: RDF Graph and SPARQL Query

Wukong. To demonstrate the generality of Pragh, We also port split live migration to Wukong [49], a state-of-the-art graph-based RDF store that supports concurrent SPARQL query processing. Pragh can

Benchmarks and workloads. We use the Lehigh University Benchmark (LUBM) [2] which is widely used to evaluate RDF query systems performance [60, 35, 27, 49]. More specifically, we use LUBM-10240 dataset where each machine deploys about 32GB data. We use the query set published in Atre et al. [7] and use a mixture workload consisting of 6 classes which is the same as original Wukong.The workload is skewed such that the starting vertices are chosen following a Zipf distribution with $\theta = 0.99$ over all vertices.

Throughput. As shown in Fig. 15, Pragh with location cache can outperform all other counterparts by up to 2.53X thanks to split live migration. Shard-based live migration only improves the mixed query throughput by about 5%, since it is hard to balance requirements for keys in each individual shard. The basic split migration outperforms shard-based migration by 1.52X (1.60X vs. 1.05X) due to more fine-grained migration. After enabling location cache, the throughput of Wukong+Pragh will further increase by 1.58X.

7 Related Work

Live migration on relational stores. There have been many efforts to provide live migration features for distributed relational databases, considering different low-level architectures, such as shared-storage [19, 20, 25, 45, 18, 9] or partitioned database [50, 24, 57, 34]. They mainly focus on migrating shards efficiently across machines for balancing load and reducing latency. There are two main types of approaches: pre-copy based [19, 20, 57] and post-copy based [25, 24, 34].

To the best of our knowledge, almost all such systems adopt shard-based mechanisms (e.g., range or hash partitioning [16, 40]) and the changes of the ownership of shard are necessary when migration. Hence, they must maintain the state of shards explicitly by using internal global data structures or external location services [52, 3, 4]. Differently, split live migration fixes the (logical) location of data to avoid the maintenance overhead, which make it different from all of previous approaches.

The inherent drawback of one-off sharding has driven a few recent efforts to support dynamic sharding [24], auto sharding [3] and application-specific sharding [4] techniques. However, when shards still serve as the unit of migration, it is hard to balance the effectiveness (granularity) and efficiency (CPU and memory) for large-scale graph data with dynamic workloads due to lacks of locality.

Live migration on graph stores. The increasing importance of graph data models has stimulated a few recent designs of vertex migration or graph repartitioning techniques targeting graph systems [41, 59, 33, 38, 62], since it is hard or even impossible to handle dynamic workloads or evolving graphs only relying on static partition-based approaches [26, 12]. Mizan [33] leverages fine-grained vertex migration to improve load balance for graph analytics workloads (e.g., PageRank). However, vertex migration can only happen when all worker threads reach a synchronization barrier (*non-live*). Further, Mizan uses a distributed hash table (DHT [8]) to locate vertices (ownership), which would still incur high memory pressure when facing large-scale graphs and additional synchronization cost.

Most graph re-partitioning approaches [41, 59, 62] need to maintain a global metadata to map vertices to partitions, and use multiple phases to iteratively migrate vertices for reducing the communication cost. Therefore, these design choices make them slow to react to changes of workloads and to other real-time events. Pragh can provide instant response to migration demands using lightweight monitoring and unilateral migration protocol.

Further, data replication has been used to improve the locality of traversal workloads over graph stores [28, 59, 37]. It will consume more memory and complicate the design of graph stores in the face of evolving graphs. Data replication is orthogonal to live migration, and how to integrated with it is part of our future work.

8 Conclusion

This paper proposes Pragh, an efficient locality-preserving live graph migration scheme for graph store. The key idea of Pragh is *split live migration*, which allows fine-grained migration while avoiding the need to maintain excessive metadata. Evaluations using both micro-benchmark and SPARQL workloads confirms the effectiveness and generality of Pragh.

References

[1] IEEE 1588 Precision Time Protocol (PTP) Version 2. http://sourceforge.net/p/ptpd/wiki/Home/.

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

[3] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *OSDI*, pages 739–753, 2016.

[4] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the shards: managing datastore locality at scale with akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 445–460, 2018.

[5] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[7] 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, New York, NY, USA, 2010. ACM.

[8] H. Balakrishnan, M. F. Kaashoek, D. Karger, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, 2003.

[9] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy. "cut me some slack": Latency-aware live migration for databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 432–443, New York, NY, USA, 2012. ACM.

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

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[12] 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, New York, NY, USA, 2015. ACM.

[13] 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.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154. ACM, 2010.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[16] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 313–324, New York, NY, USA, 2011. ACM.

[17] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, 6(11):1150–1161, Aug. 2013.

[18] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, Apr. 2013.

[19] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. *CS, UCSB, Santa Barbara, CA, USA, Tech. Rep*, 9:2010, 2010.

[20] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[21] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414. USENIX Association, 2014.

[22] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15, pages 54–70, New York, NY, USA, 2015. ACM.

[23] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *Proc. VLDB Endow.*, 9(11):852–863, July 2016.

[24] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[25] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[27] 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, New York, NY, USA, 2014. ACM.

[28] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal*, 25(3):355–380, June 2016.

[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, pages 11–11. USENIX Association, 2010.

[30] B. Iordanov. Hypergraphdb: A generalized graph database. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 25–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM'14, pages 295–306. ACM, 2014.

[32] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 485–500, Santa Clara, CA, Mar. 2016. USENIX Association.

[33] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[34] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

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

[36] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114. USENIX Association, 2013.

[37] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 145–156, New York, NY, USA, 2012. ACM.

[38] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015.

[39] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[40] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, New York, NY, USA, 2012. ACM.

[41] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 375–386, New York, NY, USA, 2010. ACM.

[42] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, Aug. 2018.

[43] L. Rietveld, R. Hoekstra, S. Schlobach, and C. Guéret. Structural properties as proxy for semantic relevance in rdf graph sampling. In *International Semantic Web Conference*, pages 81–96. Springer, 2014.

[44] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, Dec. 2017.

[45] O. Schiller, N. Cipriani, and B. Mitschang. Prorea: Live database migration for multi-tenant rdbms with snapshot isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 53–64, New York, NY, USA, 2013. ACM.

[46] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[47] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, Nov. 2016.

[48] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[49] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proc. OSDI*, 2016.

[50] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, Nov. 2014.

[51] Titan. Titan Data Model. http://s3.thinkaurelius.com/docs/titan/current/data-model.html, 2018.

[52] N. Tran, M. K. Aguilera, and M. Balakrishnan. Online migration for geo-distributed storage systems. In *USENIX Annual Technical Conference*, 2011.

[53] 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.

[54] S. Wang, C. Lou, R. Chen, and H. Chen. Fast and concurrent rdf queries using rdma-assisted gpu graph exploration. In *Proc. USENIX ATC*, 2018.

[55] 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.

[56] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

[57] X. Wei, S. Shen, R. Chen, and H. Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *Proceedings of the 2017 USENIX Annual Technical Conference*, USENIX ATC'17, pages 335–347, Santa Clara, CA, 2017. USENIX Association.

[58] 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.

[59] 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.

[60] 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. VLDB Endowment, 2013.

[61] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proc. SOSP*, 2017.

[62] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 121–132. IEEE, 2016.