

On Smart Query Routing: For Distributed Graph Querying with Decoupled Storage

Arijit Khan, ***Nanyang Technological University, Singapore;*** **Gustavo Segovia,** ***ETH Zurich, Switzerland;*** **Donald Kossmann,** ***Microsoft Research, Redmond, USA***

https://www.usenix.org/conference/atc18/presentation/khan

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

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

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

On Smart Query Routing: For Distributed Graph Querying with Decoupled Storage

Arijit Khan
NTU Singapore

Gustavo Segovia
ETH Zurich, Switzerland

Donald Kossmann
Microsoft Research, Redmond, USA

Abstract

We study online graph queries that retrieve nearby nodes of a query node in a large network. To answer such queries with high throughput and low latency, we partition the graph and process in parallel across a cluster of servers. Existing distributed graph systems place each partition on a separate server, where query answering over that partition takes place. This design has two major disadvantages. First, the router maintains a fixed routing table (or, policy), thus less flexible for query routing, fault tolerance, and graph updates. Second, the graph must be partitioned so that the workload across servers is balanced, and the inter-machine communication is minimized. To maintain good-quality partitions, it is also required to update the existing partitions based on workload changes. However, graph partitioning, online monitoring of workloads, and dynamically updating the partitions are expensive.

We mitigate these problems by decoupling graph storage from query processors, and by developing smart routing strategies with graph landmarks and embedding. Since a query processor is no longer assigned any fixed part of the graph, it is equally capable of handling any request, thus facilitating load balancing and fault tolerance. Moreover, due to our smart routing strategies, query processors can effectively leverage their cache, reducing the impact of how the graph is partitioned across storage servers. Our experiments with several real-world, large graphs demonstrate that the proposed framework gRouting, even with simple hash partitioning, achieves up to an order of magnitude better query throughput compared to existing distributed graph systems that employ expensive graph partitioning and re-partitioning strategies.

1 INTRODUCTION

Graphs with millions of nodes and billions of edges are ubiquitous to represent highly interconnected structures including the World Wide Web, social networks, knowledge graphs, genome and scientific databases, medical and government records. To support online search and query services (possibly from many clients) with low latency and high throughput, data centers and cloud operators consider scale-out solutions, in which the graph and its data are partitioned horizontally across cheap commodity servers. We assume that the graph topology and the data associated with nodes and edges are co-located, since they are often accessed together [34, 16, 17]. Keeping with the modern database trends to support low-latency operations, we target a fully in-memory system, and use disks only for durability [35, 26, 28]. In this paper, we study online queries that explore a small region of the entire graph, and require fast response time. These queries usually start with a query node, and traverse its neighboring nodes up to a certain number of hops (we shall formally introduce our queries in Section 2). For efficiently answering online queries in a distributed environment, state-of-the-art systems (e.g., [35, 28, 26]) first partition the graph, and then place each partition on a separate server, where query answering over that partition takes place (Figure 1). Since the server which contains the query node can only handle that request, the router maintains a fixed routing table (or, a fixed routing strategy, e.g., modulo hashing). Hence, these systems are less flexible with respect to query routing and fault tol-

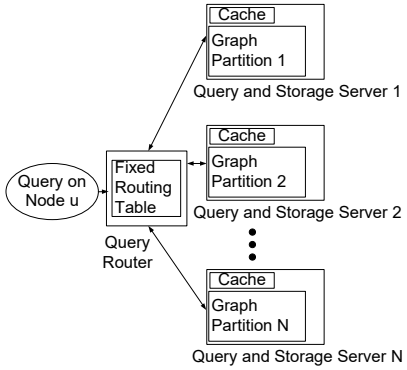


Figure 1: State-of-the-art distributed graph querying systems (e.g., SEDGE [35], Trinity [28], Horton [26])

erance, e.g., adding more machines will require updating the routing table. Besides, an effective graph partitioning in these systems must achieve: (1) workload balancing to maximize parallelism, and (2) locality of data access to minimize network communication. It has been demonstrated [35] that sophisticated partitioning schemes improve the performance of graph querying, compared to an inexpensive hash partitioning.

Due to power-law degree distribution of real-world graphs, it is difficult to get high-quality partitions [6]. Besides, a one-time partitioning cannot cope with later updates to graph structure or variations in query workloads. Several graph re- partitioning and replication-based strategies were proposed, e.g., [35, 18, 16]. However, online monitoring of workload changes, re-partitioning of the graph topology, and migration of graph data across servers are expensive; and they reduce the efficiency and throughput of online querying [25].

Our Contribution. In contrast to existing systems, we consider a different architecture, which relies *less* on an effective graph partitioning. Instead, we decouple query processing and graph storage into two separate tiers (Figure 2). In a decoupled framework, the graph is partitioned across servers allocated to the storage tier, and these storage servers hold the graph data in their main memory. Since a query processor is no longer assigned any fixed part of the graph, it is equally capable of handling any request, thus facilitating load balancing and fault tolerance. At the same time, the query router can send a request to any of the query processors, which adds more flexibility to query routing, e.g., more query processors can be added (or, a query processor that is down can be replaced) without affecting the routing strategy. Another benefit due to decoupled design is that each tier can be scaled-up independently. If a certain workload is processing intensive, more servers could be allocated to the processing tier. On the contrary, if the graph size increases over time, more servers can be added in the storage tier. This decoupled architecture, being generic, can be employed in many existing graph querying systems.

The idea of decoupling, though effective, is not novel. Facebook implemented a fast caching layer, Memcached on top of a graph database that scales the performance of graph query answering [19]. Google's F1 [29] and ScaleDB (http://scaledb.com/pdfs/TechnicalOverview.pdf) are based on a decoupling principle for scalability. Recently, Loesing et. al. [14] and Binnig et. al. [3] demonstrated the benefits of a decoupled, shared-data architecture, together with low latency and high throughput Infiniband network. Shalita et. al. [27] employed de-coupling for an optimal assignment of HTTP requests over a distributed graph storage.

Our contribution lies in designing a smart query routing logic to utilize the cache of query processors over such decoupled architecture. Achieving more cache hits is critical in a decoupled architecture – otherwise, the query processors need to retrieve the data from storage servers, which will incur extra communication costs. This is a non-trivial problem, e.g., exploiting cache locality and balancing workloads are conflicting in nature. For example, to achieve maximum cache locality, the router can send all the queries to the same processor (assuming no cache eviction happens). However, the workload of the processors will be highly imbalanced, resulting in lower throughput. In addition, graph workloads are significantly different from traditional database applications. The interconnected nature of graph data results in poor locality, and each query usually accesses multiple neighboring nodes spreading across the distributed storage. Therefore, to maximize cache hit rates at query processors, it is not sufficient to only route the queries on same nodes to the same processor. Rather, successive queries on *neighboring* nodes should also be routed to the same processor, since the neighborhoods of two nearby nodes may significantly overlap. To the best of our knowledge, such smart query routing schemes for effectively leveraging the cache contents were not considered in existing graph querying systems.

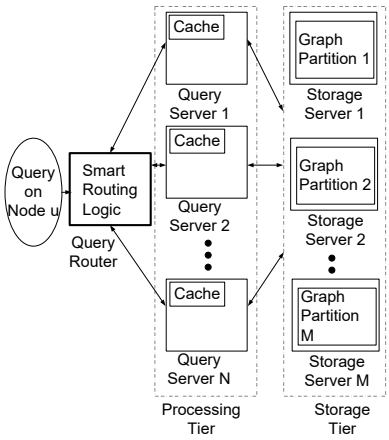


Figure 2: Decoupled architecture for graph querying

We summarize our contributions as follows.

1. We study for the first time the problem of *smart query routing* aimed at improving the throughput and efficiency of distributed graph querying.
2. In contrast to many distributed graph querying systems [35, 28, 26], we consider a different architecture that *decouples* query processors from storage layer, thereby achieving flexibility in system deployment, query routing, scaling up, load balancing, and fault tolerance.
3. We develop smart, lightweight, and adaptive *query routing algorithms* that improve cache hit rates at query processors, thus reducing communication

with storage layer, and making our design less reliant on a sophisticated graph partitioning scheme across storage layer.

4. We empirically demonstrate throughput and efficiency of our framework, gRouting on three real-life graphs, while also comparing with two existing distributed graph processing systems (SEDGE/Giraph [35] and PowerGraph [6]). Our decoupled implementation, even with its simple hash partitioning, achieves up to an order of magnitude higher throughput compared to existing systems with expensive graph partitioning schemes.

2 PRELIMINARIES

2.1 Graph Data Model

A heterogeneous network can be modeled as a labeled, directed graph $G = (V, E, \mathscr{L})$ with node set V, edge set E, and label set $\mathscr{L}$, where (1) each node $u \in V$ represents an entity in the network, (2) each directed edge $e \in E$ denotes the relationship between two entities, and (3) $\mathscr{L}$ is a function which assigns to each node u and every edge e a label $\mathscr{L}(u)$ (and $\mathscr{L}(e)$, respectively) from a finite alphabet. The node labels represent the attributes of the entities, e.g., name, job, location, etc, and edge labels the type of relationships, e.g., founder, place founded, etc.

We store the graph as an adjacency list. Every node in the graph is added as an entry in the storage where the key is the node id and the value is an array of 1-hop neighbors. If the nodes and edges have labels, they are stored in the corresponding value entry. For each node, we store both incoming and outgoing edges. Both incoming and outgoing edges of a node can be important from the context of different queries. As an example, if there is an edge *founded* from *Jerry Yang* to *Yahoo!* in a knowledge graph, there also exists a reverse relation *founded_by* from *Yahoo!* to *Jerry Yang*. Such information could be useful in answering queries about *Yahoo!*.

2.2 h-Hop Traversal Queries

We discuss various h-hop queries over heterogeneous, directed graphs in the following.

1. **h-hop Neighbor Aggregation**: Count the number of h-hop neighbors of a query node.
2. **h-step Random Walk with Restart**: The query starts at a node, and runs for h-steps — at each step, jumps to one of its neighbors with equal probability, or returns to the query node with a small probability.
3. **h-hop Reachability**: Find if a given target node is reachable from a given source node within h-hops.

The aforementioned queries are often used as the basis for more complex graph operations. For example, neighborhood aggregation is critical for node labeling and classification, that is, the label of an unlabeled node could be assigned as the most frequent label which is present within its h-hop neighborhood. The h-step random walk is useful in expert finding, ranking, discovering functional modules, complexes, and pathways. Our third query can be employed in distance-constrained and label-constrained reachability search, as well as in approximate graph pattern matching queries [16].

2.3 Decoupled Design

We decouple query processing from graph storage. This decoupling happens at a *logical* level. As an example, query processors can be different physical machines than storage servers. On the other hand, the same physical machine can also run a query processing daemon, together with storing a graph partition in its main memory as a storage server. However, the logical separation between the two layers is critical in our design.

The advantages of this separation are more flexibility in query routing, system deployment, and scaling up, as well as achieving better load balancing and fault tolerance. However, we must also consider the drawbacks of having the graph storage apart. First, query processors may need to communicate with the storage tier via the network. This includes an additional penalty to the response time for answering a query. Second, it is possible that this design causes high contention rates on either the network, storage tier, or both.

To mitigate these issues, we design smart routing schemes that route queries to processors which are likely to have the relevant data in their cache, thereby reducing the communication overhead between processing and storage tiers. Below, we discuss various components of our design, including storage, processing tier, and router.

Graph Storage Tier. The storage tier holds all graph data by horizontally partitioning it across cheap commodity servers. Sophisticated graph partitioning will benefit our decoupled architecture as follows. Let us assume that the neighboring nodes can be stored in a page within the same storage server, and the granularity of transfer from storage to processing tier is a page containing several nodes. Then, we could actually ship a set of relevant nodes with a single request if the graph is partitioned well. This will reduce the number of times data are transferred between the processing and storage tier.

However, our lightweight and smart query routing techniques exploit the notion of graph landmarks [12] and embedding [36], thereby effectively utilizing the cache of query processors that stores recently used graph data. As demonstrated in our experiments, due to our smart routing, many neighbors up to 2$\sim$3-hops of a query node can be found locally in the query processors' cache. Therefore, the partitioning scheme employed across storage servers becomes less important.

Query Processing Tier. The processing tier consists of

servers where the actual query processing takes place. These servers do not communicate with each other [14]. They only receive queries from the query router, and also request graph data from the storage tier if necessary.

To reduce the amount of calls made to the storage tier, we utilize the cache of the query processors. Whenever some data is retrieved from the storage, it is saved in cache, so that the same request can be avoided in the near future. However, it imposes a constraint on the maximum storage capacity. When the addition of a new entry surpasses this storage limit, one or more old entries are evicted from the cache. We select the LRU (i.e., Least Recently Used) eviction policy because of its simplicity. LRU is usually implemented as the default cache replacement policy, and it favors recent queries. Thus, it performs well with our smart routing schemes.

Query Router. The router creates a thread for each processor, and opens a connection to send queries by following the routing schemes which we shall describe next.

3 QUERY ROUTING STRATEGIES

When a query arrives at the router, the router decides the appropriate query processor to which the request could be sent. For existing graph querying systems, e.g., SEDGE [35] and Horton [26], where each query processor is assigned a graph partition, this decision is fixed and defined in the routing table; the processor which contains the query node handles the request. With a decoupled architecture, no such mapping exists. Hence, we design novel routing schemes with the following objectives.

3.1 Routing Algorithm Objectives

1. Leverage each processor's cached data. Let us consider t successive queries received by the router. The router will send them to query processors in a way such that the average number of *cache hits* at the processors is maximized. This, in turn, reduces the average query processing latency. However, as stated earlier, to achieve maximum cache hits, it will not be sufficient to only route the queries on same nodes to the same processor. Rather, successive queries on *neighboring* nodes should also be routed to the same processor, since the neighborhoods of two nearby nodes may significantly overlap. This will be discussed shortly in Requirement 1.

2. Balance workload even if skewed or contains hotspot. As earlier, let us consider a set of t successive queries. A naïve approach will be to ensure that each query processor receives equal number of queries, e.g., a round-robin way of query dispatching by the router. However, each query might have a different workload, and would require a different processing time. We, therefore, aim at maximizing the overall *throughput* via query stealing (explained in Requirement 2), which automatically balances the workload across query processors.

3. Make fast routing decisions. The average time at the router to dispatch a query should be minimized, ideally a small constant time, or much smaller than $\mathcal{O}(n)$, where n is the number of nodes in the input graph. This reduces the query processing latency.

4. Have low storage overhead in the router. The router may store auxiliary data to enable fast routing decisions. However, this additional storage overhead must be a small fraction compared to the graph size.

3.2 Challenges in Query Routing

It is important to note that our routing objectives are not in harmony; in fact, they are often conflicting with each other. First, in order to achieve maximum cache locality, the router can send all the queries to the same processor (assuming no cache eviction happens). However, the workload of the processors will be highly imbalanced in this case, resulting in lower throughput. Second, the router could inspect the cache of each processor before making a good routing decision, but this will add network communication delay. Hence, the router must *infer* what is likely to be in each processor's cache.

In the following, we introduce two concepts that are directly related to our routing objectives, and will be useful in designing smart routing algorithms.

Topology-Aware Locality. To understand the notion of cache locality for graph queries (i.e., routing objective 1), we define a concept called *topology-aware locality*. If u and v are nearby nodes, then successive queries on u and v must be sent to the same processor. It is very likely that the h-hop neighborhoods of u and v significantly overlap.

But, how will the router know that u and v are nearby nodes? One option is to store the entire graph topology in the router; but this could have a high storage overhead. For example, the *WebGraph* dataset that we experimented with has a topology of size 60GB. Ideally, a graph with 10^7 nodes can have up to 10^{14} edges, and in such cases, storing only the topology itself requires petabytes of memory. Thus, we impose a requirement on our smart routing schemes as follows.

Requirement 1 *The additional storage at the router for enabling smart routing should not be asymptotically larger than $\mathcal{O}(n)$, n being the number of nodes; however, the routing schemes should still be able to exploit topology-aware locality.*

Achieving this goal is non-trivial, as the topology size can be $\mathcal{O}(n^2)$, and we provision for only $\mathcal{O}(n)$ space to approximately preserve such information.

Query Stealing. Routing queries to processors that have the most useful cache data might not always be the best strategy. Due to power-law degree distribution of real-world graphs, processing queries on different nodes might require different amount of time. Therefore, the processors dealing with high-degree nodes will

have more workloads. Load imbalance can also happen if queries are concentrated in one specific region of the graph. When that happens, all queries will be sent to one processor, while other processors remain idle. To rectify such scenarios, we implement *query stealing* in our routing schemes as stated next.

Requirement 2 *Whenever a processor is idle and is ready to handle a new query, if it does not have any other requests assigned to it, it may "steal" a request that was originally intended for another processor.*

Query stealing is a well established technique for load balancing that is prevalently used by the HPC community, and there are several ways how one can implement it. *We perform query stealing at the router level*. In particular, the router sends the next query to a processor only when it receives an acknowledgement for the previous query from that processor. The router also keeps a *queue* for each connection in order to store the future queries that need to be delivered to the corresponding processor. By monitoring the length of these queues, it can estimate how busy a processor is, and this enables the router to rearrange the future queries for load balancing. We demonstrate the effectiveness of query stealing in our experiments (Section 4.6).

We next design four routing schemes — the first two are naïve and do not meet all the objectives of smart routing. On the other hand, the last two algorithms follow the requirements of a smart routing strategy.

3.3 Baseline Methods

3.3.1 Next Ready Routing

Next Ready routing is our first baseline strategy. The router decides where to send a query by choosing the next processor that has finished computing and is ready for a new request. The main advantages are: (1) It is easy to implement. (2) Routing decisions are made in constant time. (3) No preprocessing or storage overhead is required. (4) The workload is well balanced. However, this scheme fails to leverage processors' cache.

3.3.2 Hash Routing

The second routing scheme that we implement is *hash*, and it also serves as a baseline to compare against our smart routing techniques. The router applies a fixed hash function on each query node's id to determine the processor where it sends the request. In our implementation, we apply a modulo hash function.

In order to facilitate load balancing in the presence of workload skew, we implement *query stealing* mechanism. Whenever a processor is idle and is ready to handle a new query, if it does not have any other requests assigned to it, it steals a request that was originally intended for another processor. Since queries are queued in the router, the router is able to take this decision, and ensures that there are no idle processors when there is still some work to be done. Our hash routing has all the benefits of next ready, and very likely it sends a repeated query to the same processor, thereby getting better locality out of the cache. However, hash routing cannot capture topology-aware locality.

3.4 Proposed Methods

3.4.1 Landmark Routing

Our first smart routing scheme is based on landmark nodes [12]. One may recall that we store both incoming and outgoing edges of every node, thus we consider a *bi-directed* version of the input graph in our smart routing algorithms. We select a small set L of nodes as landmarks, and also pre-compute the distance of every node to these landmarks. We determine the optimal number of landmarks based on empirical results. Given some landmark node $l \in L$, the distance $d(u,v)$ between any two nodes u and v are bounded as follows:

$$|d(u,l) - d(l,v)| \leq d(u,v) \leq d(u,l) + d(l,v) \qquad (1)$$

Intuitively, if two nodes are close to a given landmark, they are likely to be close themselves. Our landmark routing is based on the above principle. We first select a set of landmarks that partitions the graph into P regions, where P is the total number of processors. We then decide a one-to-one mapping between those regions and processors. Now, if a query belongs to a specific region (decided based on its distance to landmarks), it is routed to the corresponding processor. Clearly, this routing strategy requires a preprocessing phase as follows.

Preprocessing. We select landmarks based on their node degree and how well they spread over the graph [1]. Our first step is to find a certain number of landmarks considering the highest degree nodes, and then compute their distance to every node in the graph by performing breadth first searches (BFS). If we find two landmarks to be closer than a pre-defined threshold, the one with the lower degree is discarded. The complexity of this step is $\mathcal{O}(|L|e)$, due to $|L|$ number of BFS, where $|L|$ is the number of landmarks, and e is the number of edges.

Next, we assign the landmarks to query processors as follows. First, every processor is assigned a "pivot" landmark with the intent that pivot landmarks are as far from each other as possible. The first two pivot landmarks are the two that are farthest apart considering all other landmark pairs. Each next pivot is selected as the landmark that is farthest from all previously selected pivot landmarks. Each remaining landmark is assigned to the processor which contains its closest pivot landmark. The complexity of this step is $\mathcal{O}(|L|^2 + |L|P)$, where P is the number of processors.

Finally, we define a "distance" metric d between the graph nodes and query processors. The distance of a

node u to a processor p is defined as the minimum distance of u to any landmark that is assigned to processor p. This information is stored in the router, which requires $\mathcal{O}(nP)$ space and $\mathcal{O}(nL)$ time to compute, where n is the number of nodes. Therefore, *the storage requirement at the router is linear in the number of nodes*.

Routing. To decide where to send a query on node u, the router verifies the pre-computed distance $d(u,p)$ for every processor p, and selects the one with the smallest $d(u,p)$ value. As a consequence, the routing decision time is linear in the number of processors: $\mathcal{O}(P)$. This is very efficient since the number of processors is small.

In contrast to our earlier baseline routings, this method is able to leverage topology-aware locality. It is likely that query nodes that are in the neighborhood of each other will have similar distances to the processors; hence, they will be routed in a similar fashion. On the other hand, the landmark routing scheme is less flexible with respect to addition or removal of processors, since the assignment of landmarks to processors, as well as the distances $d(u,p)$ for every node u and each processor p needs to be recomputed.

The distance metric $d(u,p)$ is useful not only in finding the best processor for a certain query, but it can also be used for load balancing, fault tolerance, dealing with workload skew, and hotspots. As an example, let us assume that the closest processor for a certain query is very busy, or is currently down. Since the distance metric gives us distances to all processors, the router is able to select the second, third, or so on closest processor. This form of load balancing will impact the nearby query nodes in the same way; and therefore, the modified routing scheme will still be able to capture topology-aware locality. In practice, it can be complex to define exactly when a query should be routed to its next best query processor. We propose a formula that calculates the *load-balanced distance* $d^{LB}(u,p)$ as given below.

$$d^{LB}(u,p) = d(u,p) + \frac{\text{Processor Load}}{\text{Load Factor}} \quad (2)$$

Thus, the query is always routed to the processor with the smallest $d^{LB}(u,p)$. The router uses the number of queries in the queue corresponding to a processor as the measure of its load. The load factor is a tunable parameter, which allows us to decide how much load would result in the query to be routed to another processor. We find its optimal value empirically.

Dealing with Graph Updates. During addition/ deletion of nodes and edges, one needs to recompute the distances from every node to each of the landmarks. This can be performed efficiently by keeping an additional *shortest-path-tree* data structure [31]. However, to avoid the additional space and time complexity of maintaining a shortest-path-tree, we follow a simpler approach. When a new node u is added, we compute the distance of this node to every landmark, and also its distance $d(u,p)$ to every processor p. In case of an edge addition or deletion between two existing nodes, for these two end-nodes and their neighbors up to a certain number of hops (e.g., 2-hops), we recompute their distances to every landmark, as well as to every processor. Finally, in case of a node deletion, we handle it by considering deletion of multiple edges that are incident on it. After a significant number of updates, previously selected landmark nodes become less effective; thus, we recompute the entire preprocessing step periodically in an off-line manner.

3.4.2 Embed Routing

Our second smart routing scheme is the *Embed* routing, which is based on graph embedding [36, 4]. We embed a graph into a lower dimensional Euclidean space such that the hop-count distance between graph nodes are approximately preserved via their Euclidean distance (Figure 3). We then use the resulting node co-ordinates to determine how far a query node is from the recent history of queries that were sent to a specific processor. Clearly, embed routing also requires a preprocessing step.

Preprocessing. For efficiently embedding a large graph in a D-dimensional Euclidean plane, we first select a set L of landmarks and find their distances from each node in the graph. We then assign co-ordinates to landmark nodes such that the distance between each pair of landmarks is approximately preserved. We, in fact, minimize the *relative error* in distance for each pair of landmarks, defined below.

$$f_{error}(v_1,v_2) = \frac{|d(v_1,v_2) - \text{EuclideanDist}(v_1,v_2)|}{d(v_1,v_2)} \quad (3)$$

Here, $d(v_1,v_2)$ is the hop-count distance between v_1 and v_2 in the original graph, and $\text{EuclideanDist}(v_1,v_2)$ is their Euclidean distance after the graph is embedded. We minimize the relative error since we are more interested in preserving the distances between nearby node pairs. Our problem is to minimize the aggregate of such errors over all landmark pairs — this can be cast as a generic multi-dimensional global minimization problem, and could be approximately solved by many off-the-shelf techniques, e.g., the *Simplex Downhill* algorithm that we apply in this work. Next, every other node's co-ordinates are found also by applying the Simplex Downhill algorithm that minimizes the aggregate relative distance error between the node and all the landmarks. The overall graph embedding procedure consumes a modest preprocessing time: $\mathcal{O}(|L|e)$ due to BFS from $|L|$ landmarks, $\mathcal{O}(|L|^2D)$ for embedding the landmarks, and $\mathcal{O}(n|L|D)$ for embedding the remaining nodes. In addition, the second step is completely parallelizable per node. Since each node receives D co-ordinates, it requires total $\mathcal{O}(nD)$ space in the router, which is linear in the number of nodes. Unlike landmark routing, *a benefit*

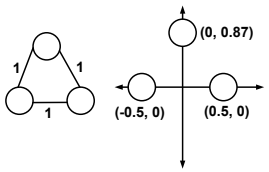


Figure 3: Example of graph embedding in 2D Euclidean plane

of embed routing is that the preprocessing is independent of the system topology, allowing more processors to be easily added at a later time.

Routing. The router has access to each node's co-ordinates. By keeping an average of the query nodes' co-ordinates that it sent to each processor, it is able to infer the cache contents in these processors. As a consequence, the router finds the distance between a query node u and a processor p, denoted as $d_1(u,p)$, and defined as the distance of the query node's co-ordinates to the historical mean of the processor's cache contents. As recent queries are more likely to influence the cache contents due to LRU eviction policy, we use the exponential moving average to compute the mean of the processor's cache contents. Initially, the mean co-ordinates for each processor are assigned uniformly at random. Next, assuming that the last query on node v was sent to processor p, its updated mean co-ordinates are:

$$\text{MeanCo-ordinates}(p) = \alpha \cdot \text{MeanCo-ordinates}(p) + (1-\alpha) \cdot \text{Co-ordinates}(v) \quad (4)$$

The smoothing parameter $\alpha \in (0,1)$ in the above Equation determines the degree of decay used to discard older queries. For example, α close to 0 assigns more weight only to the last query, and α close to 1 decreases the weight on the last query. We determine the optimal value of α based on experimental results. Finally, the distance between a query node u and a processor p is computed as given below.

$$d_1(u,p) = ||\text{MeanCo-ordinates}(p) - \text{Co-ordinates}(u)|| \quad (5)$$

Since we embed in an Euclidean plane, we use the L_2 norm to compute distances. We select the processor with the smallest $d_1(u,p)$ distance. One may observe that the routing decision time is only $\mathcal{O}(PD)$, P being the number of processors and D the number of dimensions.

Analogous to landmark routing, we now have a distance to each processor for a query; and hence, we are able to make routing decisions taking into account the processors' workloads and faults. As earlier, we define a *load-balanced distance* $d_1^{LB}(u,p)$ between a query node u and a processor p, and the query is always routed to the processor with the smallest $d_1^{LB}(u,p)$ value.

$$d_1^{LB}(u,p) = d_1(u,p) + \frac{\text{Processor Load}}{\text{Load Factor}} \quad (6)$$

The embed routing has all the benefits of smart routing. This routing scheme divides the active regions (based on workloads) of the graph into P partitions in an overlapping manner, and assigns them to the processors' cache. Moreover, it dynamically adapts the partitions with new workloads. Therefore, *it bypasses the expensive graph partitioning and re-partitioning problems to the existing cache replacement policy of the query processors*. This shows the effectiveness of embed routing.

Dataset	# Nodes	# Edges	Size on Disk (Adj. List)
WebGraph	105 896 555	3 738 733 648	60.3 GB
Memetracker	96 608 034	418 237 269	8.2 GB
Freebase	49 731 389	46 708 421	1.3 GB

Table 1: Graph datasets

Dealing with Graph Updates. Due to pre-assignment of node co-ordinates, embed routing is less flexible with respect to graph updates. When a new node is added, we compute its distance from the landmarks, and then assign co-ordinates to the node by applying the Simplex Downhill algorithm. Edge updates and node deletions are handled in a similar method as discussed for landmark routing. We recompute the entire preprocessing step periodically in an off-line manner to deal with a significant number of graph updates.

4 EVALUATION

4.1 Experiment Setup

• **Cluster Configuration.** We perform experiments on a cluster of 12 servers having 2.4 GHz Intel Xeon processors, and interconnected by 40 Gbps Infiniband, and also by 10 Gbps Ethernet. Most experiments use a single core of each server with the following configuration: 1 server as router, 7 servers in the processing tier, 4 servers in the storage tier; and communication over Infiniband with remote direct memory access (RDMA). Infiniband allows RDMA in a few microseconds. We use a limited main memory (0∼4GB) as the cache of processors. Our codes are implemented in C++.

To implement our storage tier, we use RAMCloud [20], which provides high throughput and very low read/write latency, in the order of 5-10 μs for every put/get operation. It is able to achieve this efficiency because it keeps all stored values in memory as a distributed key-value store, where a key is hashed to determine on which server the corresponding key-value pair will be stored.

• **Datasets.** We summarize our data sets in Table 1. As explained in Section 2, we store both in- and out-neighbors. The graph is stored as an adjacency list — every node-id in the graph is the key, and the corresponding value is an array of its 1-hop neighbors. The graph is partitioned across storage servers via RAMCloud's default and inexpensive hash partitioning scheme, MurmurHash3 over graph nodes.

WebGraph: The uk-2007-05 web graph (http://law.di.unimi.it/ datasets.php) is a collection of web pages, which are represented as nodes, and their hyperlinks as

edges. **Memetracker:** This dataset (snap.stanford.edu) tracks quotes and phrases that appeared from August 1 to October 31, 2008 across online news spectrum. We consider documents as nodes and hyper-links as edges. **Freebase:** We download the *Freebase* knowledge graph from http://www.freebase.com/. Nodes are named entities (e.g., Google) or abstract concepts (e.g., Asian people), and edges denote relations (e.g., founder).

• **Online Query Workloads.** We consider three online graph queries [35], discussed in Section 2.2 — all require traversals up to h hops: (1) h-hop neighbor aggregation, (2) h-step random walk with restart, and (3) h-hop reachability. We consider a uniform mixture of above queries. We simulate a scenario when queries are drawn from a hotspot region; and the hotspots change over time. In particular, we select 100 nodes from the graph uniformly at random. Then, for each of these nodes, we select 10 different query nodes which are at most r-hops away from that node. Thus, we generate 1000 queries; every 10 of them are from one hotspot region, and the pairwise distance between any two nodes from the same hotspot is at most $2r$. Finally, all queries from the same hotspot are grouped together and sent consecutively. We report our results averaged over 1000 queries.

To realize the effect of topology-aware locality, we consider smaller values of r and h, e.g., $r = 2$ and $h = 2$.

• **Evaluation Metrics.**

Query Response Time measures the average time required to answer one query.

Query Processing Throughput measures the number of queries that can be processed per unit time.

Cache Hit Rate: We report cache hit rates, since higher cache hit rates reduce the query response time. Consider t queries $q_1, q_2, \ldots, q_t$ received successively by the router. For simplicity, let us assume that each query retrieves all h-hop neighbors of that query node (i.e., h-hop neighborhood aggregation). We denote by $|N_h(q_i)|$ the number of nodes within h-hops from q_i. Among them, we assume that $|N_h^c(q_i)|$ number of nodes are found in the query processors' cache.

$$\text{Cache Hit Rates} := \sum_{i=1}^{t} |N_h^c(q_i)| \tag{7}$$

$$\text{Cache Miss Rates} := \sum_{i=1}^{t} \left(|N_h(q_i)| - |N_h^c(q_i)|\right) \tag{8}$$

• **Parameter Setting.** We find that *embed routing performs the best compared to three other routing strategies*. We also set the following parameter values since they perform the best in our implementation. We shall, however, demonstrate sensitivity of our routing algorithms with these parameters in Section 4.6.

We use maximum 4GB cache in each query processor. All experiments are performed with the cache initially empty (cold cache). The number of landmarks $|L|$ is set as 96 with at least 3 hops of separation from each other. For graph embedding, 10 dimensions are used. Load Factor (which impacts query stealing) is set as 20, and the smoothing parameter $\alpha = 0.5$.

In order to realize how our routing schemes perform when there is no cache in processors, we consider an additional "no-cache" scheme. In this mode, all queries are routed following the next ready technique; however, as there is no cache in query processors, there will be no overhead due to cache lookup and maintenance.

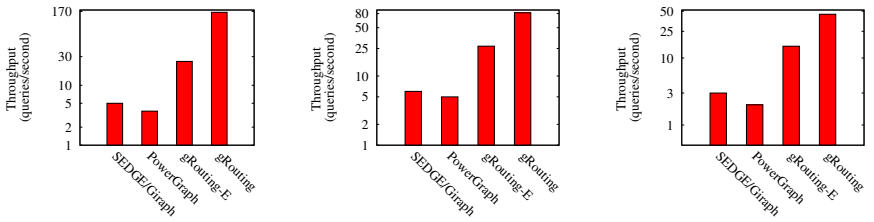
• **Compared Systems.** Decoupled architecture and our smart routing logic, being generic, can benefit many graph querying systems. Nevertheless, we compare gRouting with two distributed graph processing systems: SEDGE/Giraph [35] and PowerGraph [6]. Other recent graph querying systems, e.g., [26, 19] are not publicly available for a direct comparison.

SEDGE [35] was developed for h-hop traversal queries on top of Giraph or Google's Pregel system [15]. It follows in-memory, vertex-centric, bulk-synchronous parallel model. SEDGE employs ParMETIS software [9] for graph partitioning and re-partitioning. PowerGraph [6] follows in-memory, vertex-centric, asynchronous gather-apply-scatter model. In the beginning, only the query node is active, and each active node then activates its neighbors, until all the h-hop neighbors from the query nodes are activated. PowerGraph also employs a sophisticated node-cut based graph partitioning method.

4.2 Comparison with Graph Systems

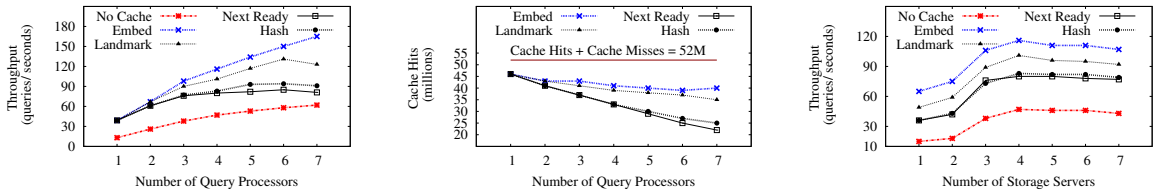
We compare gRouting (embed routing is used) with two distributed graph processing systems, SEDGE/Giraph [35] and PowerGraph [6]. As these systems run on Ethernet, we consider a version of gRouting on Ethernet (gRouting-E). We consider 12 machines configuration of SEDGE and PowerGraph, since query processing and graph storage in them are coupled on same machines. In contrast, we fix the number of routing, processing, and storage servers as 1, 7 and 4, respectively. The average 2-hop neighborhood size varies from 10K∼60K nodes over our datasets.

In Figure 4, we find that our throughput, with hash partitioning and over Ethernet, is 5∼10 times better than SEDGE and PowerGraph that employ expensive graph partitioning and re- partitioning. The re-partitioning in SEDGE requires around 1 hour and also apriori information on future queries, whereas PowerGraph graph partitioning finishes in 30 min. On the contrary, gRouting performs lightweight hash partitioning over graph nodes, and does not require any prior knowledge of the future workloads. Moreover, our throughput over Infiniband is 10∼35 higher than these systems. *These results show the usefulness of smart query routing over expensive graph partitioning and re-partitioning schemes.*



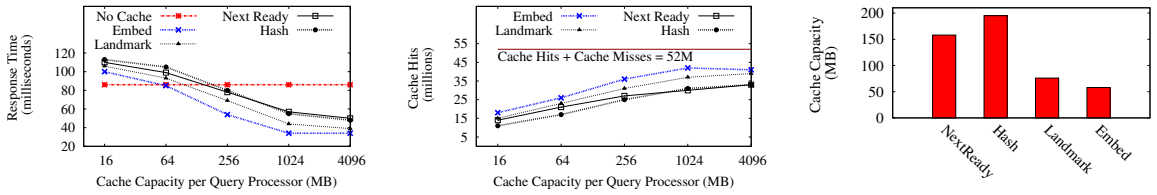
(a) *WebGraph* (b) *MemeTracker* (c) *Freebase*

Figure 4: Throughput comparison



(a) Throughput (b) Cache Hits (c) Throughput

Figure 5: Performance with varying number of query processors and storage servers, *WebGraph*



(a) Response time (b) Cache hits (c) Min. cache required to reach No-Cache's response time

Figure 6: Impact of cache size, *WebGraph*

Next, we report scalability, impact of cache sizes, and graph updates over our largest *Webgraph* dataset, and using Infiniband network.

4.3 Scalability and Deployment Flexibility

One of the main benefits of separating processing and storage tiers is deployment flexibility — they can be scaled-up independently, which we investigate below.

Processing Tier: We vary the number of processing servers from 1 to 7, while using 1 router and 4 storage servers. In Figure 5(a), we show throughput with varying number of processing servers. Corresponding cache hit rates are presented in Figure 5(b). For these experiments, we assume that each query processor has sufficient cache capacity (4GB) to store the results of all 1000 queries (i.e, adjacency lists of 52M nodes, shown in Figure 5(b)). Since, for every experiment, we start with an empty cache, and then send the same 1000 queries in order, maximum cache hit happens when there is only one query processor. As we increase the number of query processors, these queries get distributed and processed by different processors, thus cache hit rate generally decreases. This is more evident for our baseline routing schemes, and we find that their throughput saturates with 3∼5 servers. These findings demonstrate the usefulness of smart query routing: *To maintain same cache hit rate, queries must be routed intelligently. Since Embed routing is able to sustain almost same cache hit rate with many query processors (Figure 5(b)), its throughput scales linearly with query processors.*

Storage Tier: We next vary the number of storage servers from 1 to 7, whereas 1 server is used as the router and 4 servers as query processors (Figure 5(c)). When we use 1 storage server, we can still load the entire 60GB *Webgraph* on the main memory of that server, since each of our servers has sufficient RAM. The throughput is the least when there is only one storage server. We observe that 1∼2 storage servers are insufficient to handle the demand created by 4 query processors. However, with 4 storage servers, the throughput saturates, since the bottleneck is transferred to query processors. This is evident from our previous results — the throughput with 4 query processors was about 120 queries per second (Figure 5(a)), which is the same throughput achieved with 4 storage servers in the current experiments.

4.4 Impact of Cache Sizes

In previous experiments, we assign 4GB cache to each processor, which was large enough for our queries; and we never discarded anything from the cache. We next perform experiments when it needs to evict cache entries. In Figure 6, we present average response times with various cache capacities. At the largest, with 4GB cache per processor, no eviction occurs. Therefore, there is no additional performance gain by increasing the cache capacity. On the other extreme, having cache with less than

Landmark	Embed	
	embed per landmark	embed per node
35 sec	36 sec	1 sec

Table 2: Preprocess times

Landmark	Embed	Input graph
2.8 GB	4GB	60.3GB

Table 3: Preprocess storage

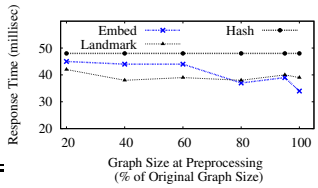
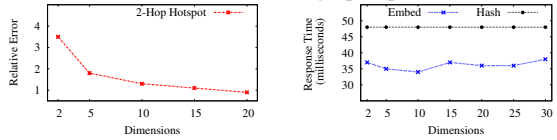


Figure 7: Robustness with graph updates



(a) Relative error (b) Response time

Figure 8: Impact of embedding dimensionality

64MB per processor results in worse response times than what was obtained with no-cache scheme, represented by the horizontal red line (86ms in Figure 6). When the cache does not have much space, it ends up evicting entries that might have been useful in the future. Hence, there are not enough cache hits to justify its maintenance and lookup costs when cache size $<$ 64MB/ processor.

We also evaluate our routing strategies in terms of minimum cache requirement to achieve a response time of 86ms, the break-even point of deciding whether or not to add a cache. Figure 6(c) shows that smart routing schemes achieve this response time with a much lower cache, as compared to that of the baselines. These results illustrate that *our smart routings utilize the cache well; and for the same amount of cache, they achieve lower response time compared to baseline routings*.

4.5 Preprocessing and Graph Updates

Preprocessing Time and Storage: For landmarks routing, we compute the distance of every node to all landmarks, which can be evaluated by performing a BFS from each landmark. This takes about 35 sec for one landmark in *Webgraph* (Table 2), and can be parallelized per landmark. For embed routing, in addition, we need to embed every node with respect to landmarks, which requires about 1 sec per node in *Webgraph*, and is again parallelizable per node.

The preprocessed landmark routing information consumes about 2.8GB storage space in case of *Webgraph*. On the contrary, with embedding dimensionality 10, the *Webgraph* embedding size is only 4GB. Both these preprocessed information are modest compared to the original *Webgraph* size, which is around 60GB (Table 3).

Graph Updates: In these experiments, we preprocess a reduced subgraph of the original dataset. For example, at 20% of the original dataset (Figure 7), we select only 20% of all nodes uniformly at random, and compute preprocessed information over the subgraph induced by these selected nodes. However, we always run our query over the complete *Webgraph*. We incrementally compute

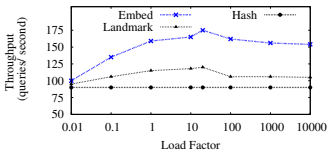


Figure 9: Impact of load factor

the necessary information for the new nodes, as they are being added, without changing anything on the preprocessed information of the earlier nodes. As an example, in case of embed routing, we only compute the distance of a new node to the landmarks, and thereby find the coordinates of that new node. However, one may note that with the addition of every new node and its adjacent edges, the preprocessed information becomes outdated (e.g., the distance between two earlier nodes might decrease). Since we do not change anything on the preprocessed information, this experiment demonstrates the robustness of our method with respect to graph updates.

Figure 7 depicts that *our smart routing schemes are robust for a small number of graph updates*. With embed routing, preprocessed information over the whole graph results in response time of 34 ms, whereas preprocessed information at 80% of the graph results in response time of 37 ms (i.e., response time increases by only 3 ms). As expected, the response time deteriorates when preprocessing is performed on a smaller amount of graph data, e.g., with only 20% graph data, response time increases to 44 ms, which is comparable to the response time of baseline hash routing (48 ms).

4.6 Sensitivity Analysis

We find that gRouting is more sensitive towards load factor (due to *query stealing*) and embedding dimensionality, compared to other parameters, e.g., number of landmarks and smoothing factor (α). Due to lack of space, we present sensitivity analysis with respect to load factor and embedding dimensionality in Figures 8 and 9. Sensitivity results with other parameters can be found in our extended version [10]. In all figures, we also show our best baseline — hash routing, for comparison.

Embedding Dimensionality: We consider the performance implications of the number of dimensions on embed routing. For these experiments, we create several embeddings, with dimensionality from 2 to 30. While the relative error in distance between node pairs decreases with higher dimensions, it almost saturates after 10 dimensions (Figure 8(a)). On the other hand, we observe that the average response time reduces until dimension 10, and then it slowly increases with more dimensions (Figure 8(b)). This is because with higher dimensions, we reduce the distance prediction error, thereby correctly routing the queries and getting more cache hits. However, a large number of dimensions also increases the routing decision making time at the router. Hence, the least response time is achieved at dimensionality 10.

Load Factor: This parameter impacts both of our smart routing schemes. We find from Equations 3 and 7 that smaller values of load factor diminish the impact of "smart" routing (i.e., landmarks and node co-ordinates), instead queries will be routed to the processor having the minimum workload. On the other hand, higher values of load factor reduces the impact of load balancing (i.e., query stealing) — queries would be routed solely based on landmarks and node co-ordinates. Therefore, in these experiments, we expect that the throughput will initially increase with higher values of load factor, until it reaches a maximum, and then it would start decreasing. Indeed, it can be observed in Figure 9 that with load factor between 10∼20, the best throughput is achieved.

5 Related Work

We studied *smart query routing* for distributed graph querying — a problem for which we are not aware of any prior work. In the following we, however, provide a brief overview of work in neighborhood areas.

Landmarks and Graph Embedding. Landmarks were used in path finding, shortest path estimation, and in estimating network properties [12, 24, 1, 23]. Graph embedding [36] was employed in internet routing, such as predicting internet network distances and estimating minimum round trip time between hosts [4]. To the best of our knowledge, ours is the first study that applies graph embedding and landmarks to design effective routing algorithms for distributed graph querying.

Graph Partitioning, Re-partitioning, Replication. The balanced, minimum-edge-cut graph partitioning divides a graph into k partitions such that each partition contains same number of nodes, and the number of cut-edges is minimized. Even for $k = 2$, the problem is **NP**-hard, and there is no approximation algorithm with a constant approximation ratio unless **P** =**NP** [18]. Therefore, efforts were made in developing polynomial-time heuristics — METIS, Chaco, SCOTCH, to name a few. More sophisticated graph partitioning schemes were also proposed, e.g., node-cut [6], complementary partitioning [35], and label propagation [33], among many others.

Graph re-partitioning is critical for online queries, since the graph topology and workload change over time [16]. The methods in [35, 18, 11] perform re-partitioning based on past workloads. Incremental partitioning was developed for dynamic and stream graphs [37, 32, 30]. With the proposed embed routing, we bypass these expensive graph partitioning and re-partitioning challenges to the existing cache replacement policy.

Replication was used for graph partitioning, re-partitioning, load balancing, and fault tolerance. In earlier works, [22, 8] proposed one extreme version by replicating the graph sufficiently so that, for every node in the graph, all of its neighbors are present locally. Mondal et. al. designed an overlapping graph re-partitioning scheme [16], which updates its partitions based on the past read/ write patterns. Huang et. al. [7] designed a lightweight re- partitioning and replication scheme considering access locality, fault tolerance, and dynamic updates. While we also replicate the graph data at query processors' cache in an overlapping manner, we *only replicate the active regions* of the graph based on recent workloads. Unlike [16, 7] we do not explicitly run any graph replication strategy at our processors or storage servers. Instead, our *smart routing algorithms* automatically perform replications at processors' cache.

Graph Caching, De-coupling, Multi-Query Optimization. Facebook uses a fast caching layer, Memcached on top of a graph database to scale the performance of graph querying [19]. Graph-structure-aware and workload-adaptive caching techniques were also proposed, e.g., [2, 21]. There are other works on view-based graph query answering [5] and multi-query optimizations [13]. Unlike ours, these approaches require the workload to be known in advance.

Recently, Shalita et. al. [27] employed decoupling for an optimal assignment of HTTP requests over a distributed graph storage. First, they perform a static partition of the graph in storage servers based on co-access patterns. Next, they find past workloads on each partition, and dynamically assign these partitions to query processors such that load balancing can be achieved. While their decoupling principle and dynamic assignment at query processors are similar to ours, they still explicitly perform a sophisticated graph partitioning at storage servers, and update such partitions in an offline manner. In contrast, our smart routing algorithms automatically partition the active regions of the graph in a dynamic manner and store them in the query processors' cache, thereby achieving both load balancing and improved cache hit rates.

6 CONCLUSIONS

We studied h-hop traversal queries – a generalized form of various online graph queries that access a small region of the graph, and require fast response time. To answer such queries with low latency and high throughput, we follow the principle of decoupling query processors from graph storage. Our work emphasized *less* on the requirements for an expensive graph partitioning and re-partitioning technique, instead we developed smart query routing strategies for effectively leveraging the query processors' cache contents, thereby improving the throughput and reducing latency of distributed graph querying. In addition to workload balancing and deployment flexibility, gRouting is able to provide linear scalability in throughput with more number of query processors, works well in the presence of query hotspots, and is also adaptive to workload changes and graph updates.

7 Acknowledgement

The research is supported by MOE Tier-1 RG83/16 and NTU M4081678. Any opinions, findings, and conclusions in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] AKIBA, T., IWATA, Y., AND YOSHIDA, Y. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *SIGMOD* (2013).

[2] AKSU, H., CANIM, M., CHANG, Y., KORPEOGLU, I., AND ULUSOY, Ö. Graph Aware Caching Policy for Distributed Graph Stores. In *IC2E* (2015).

[3] BINNIG, C., CROTTY, A., GALAKATOS, A., KRASKA, T., AND ZAMANIAN, E. The End of Slow Networks: It's Time for a Redesign. *PVLDB 9*, 7 (2016), 528–539.

[4] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM* (2004).

[5] FAN, W., WANG, X., AND WU, Y. Answering Graph Pattern Queries using Views. In *ICDE* (2014).

[6] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI* (2012).

[7] HUANG, J., AND ABADI, D. J. Leopard: Lightweight Edge-oriented Partitioning and Replication for Dynamic Graphs. *PVLDB 9*, 7 (2016), 540–551.

[8] HUANG, J., ABADI, D. J., AND REN, K. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB 4*, 11 (2011), 1123–1134.

[9] KARYPIS, G. METIS and ParMETIS. In *Encyclopedia of Parallel Computing*. Springer, 2011.

[10] KHAN, A., SEGOVIA, G., AND KOSSMANN, D. On Smart Query Routing: For Distributed Graph Querying with Decoupled Storage. `https://arxiv.org/abs/1611.03959`, 2016.

[11] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *EuroSys* (2013).

[12] KLEINBERG, J., SLIVKINS, A., AND WEXLER, T. Triangulation and Embedding Using Small Sets of Beacons. *J. ACM 56*, 6 (2009), 32:1–32:37.

[13] LE, W., KEMENTSIETSIDIS, A., DUAN, S., AND LI, F. Scalable Multi-query Optimization for SPARQL. In *ICDE* (2012).

[14] LOESING, S., PILMAN, M., ETTER, T., AND KOSSMANN, D. On the Design and Scalability of Distributed Shared-Data Databases. In *SIGMOD* (2015).

[15] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-scale Graph Processing. In *SIGMOD* (2010).

[16] MONDAL, J., AND DESHPANDE, A. Managing Large Dynamic Graphs Efficiently. In *SIGMOD* (2012).

[17] MONDAL, J., AND DESHPANDE, A. EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs. In *SIGMOD* (2014).

[18] NICOARA, D., KAMALI, S., DAUDJEE, K., AND CHEN, L. Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. In *EDBT* (2015).

[19] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *NSDI* (2013).

[20] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev. 43*, 4 (2010), 92–105.

[21] PAPAILIOU, N., TSOUMAKOS, D., KARRAS, P., AND KOZIRIS, N. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *SIGMOD* (2015).

[22] PUJOL, J. M., ERRAMILLI, V., SIGANOS, G., YANG, X., LAOUTARIS, N., CHHABRA, P., AND RODRIGUEZ, P. The Little Engine(s) That Could: Scaling Online Social Networks. In *SIGCOMM* (2010).

[23] QIAO, M., CHENG, H., CHANG, L., AND YU, J. X. Approximate Shortest Distance Computing: A Query-Dependent Local Landmark Scheme. In *ICDE* (2012).

[24] RATTIGAN, M. J., MAIER, M. E., AND JENSEN, D. Using Structure Indices for Efficient Approximation of Network Properties. In *KDD* (2006).

[25] ROY, A., BINDSCHAELDER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP* (2015).

[26] SARWAT, M., ELNIKETY, S., HE, Y., AND MOKBEL, M. F. Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs. *PVLDB 6*, 14 (2013), 1918–1929.

[27] SHALITA, A., KARRER, B., KABILJO, I., SHARMA, A., PRESTA, A., ADCOCK, A., KLLAPI, H., AND STUMM, M. Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks. In *NSDI* (2016).

[28] SHAO, B., WANG, H., AND LI, Y. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD* (2013).

[29] SHUTE, J., VINGRALEK, R., SAMWEL, B., HANDY, B., WHIPKEY, C., ROLLINS, E., OANCEA, M., LITTLEFIELD, K., MENESTRINA, D., ELLNER, S., CIESLEWICZ, J., RAE, I., STANCESCU, T., AND APTE, H. F1: A Distributed SQL Database That Scales. *PVLDB 6*, 11 (2013), 1068–1079.

[30] STANTON, I., AND KLIOT, G. Streaming Graph Partitioning for Large Distributed Graphs. In *KDD* (2012).

[31] TRETYAKOV, K., A.-CERVANTES, A., G.-BANUELOS, L., VILO, J., AND DUMAS, M. Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs. In *CIKM* (2011).

[32] VAQUERO, L., CUADRADO, F., LOGOTHETIS, D., AND MARTELLA, C. Adaptive Partitioning for Large-scale Dynamic Graphs. In *SOCC* (2013).

[33] WANG, L., XIAO, Y., SHAO, B., AND WANG, H. How to Partition a Billion-Node Graph. In *ICDE* (2014).

[34] WEI, J., XIA, F., SHA, C., XU, C., HE, X., AND ZHOU, A. Workload-Aware Cache for Social Media Data. In *APWeb* (2013).

[35] YANG, S., YAN, X., ZONG, B., AND KHAN, A. Towards Effective Partition Management for Large Graphs. In *SIGMOD* (2012).

[36] ZHAO, X., SALA, A., WILSON, C., ZHENG, H., AND ZHAO, B. Y. Orion: Shortest Path Estimation for Large Social Graphs. In *WOSN* (2010).

[37] ZHENG, A., LABRINIDIS, A., AND CHRYSANTHIS, P. K. Planar: Parallel Lightweight Architecture-Aware Adaptive Graph Re-partitioning. In *ICDE* (2016), pp. 121–132.